Stefan Sokołowski

JĘZYKI FORMALNE I METODY KOMPILACJI

wykład 14

Inst. Informatyki Stosowanej, ANS
Elbląg, 2024/2025

Działanie BISONa — kompilacja

Przykład: „języczek”

⟨program⟩ ::= ⟨komendy⟩ .

⟨komendy⟩ ::= (puste)

| ⟨komendy⟩ ⟨komenda⟩

⟨komenda⟩ ::= ⟨zmienna⟩ = ⟨zmienna⟩ ;

| ⟨zmienna⟩ = ⟨zmienna⟩ + ⟨zmienna⟩ ;

| ⟨zmienna⟩ = ⟨zmienna⟩ − ⟨zmienna⟩ ;

| ⟨zmienna⟩ = ⟨liczba⟩ ;

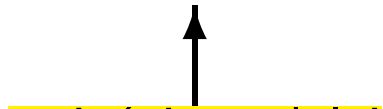
| ⟨zmienna⟩ = ⟨zmienna⟩ + ⟨liczba⟩ ;

| ⟨zmienna⟩ = ⟨zmienna⟩ − ⟨liczba⟩ ;

| (⟨zmienna⟩) { ⟨komendy⟩ } (pętla)

⟨liczba⟩ ::= ...

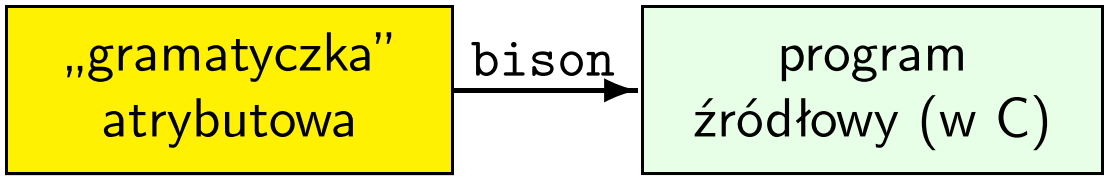
⟨zmienna⟩ ::= ...



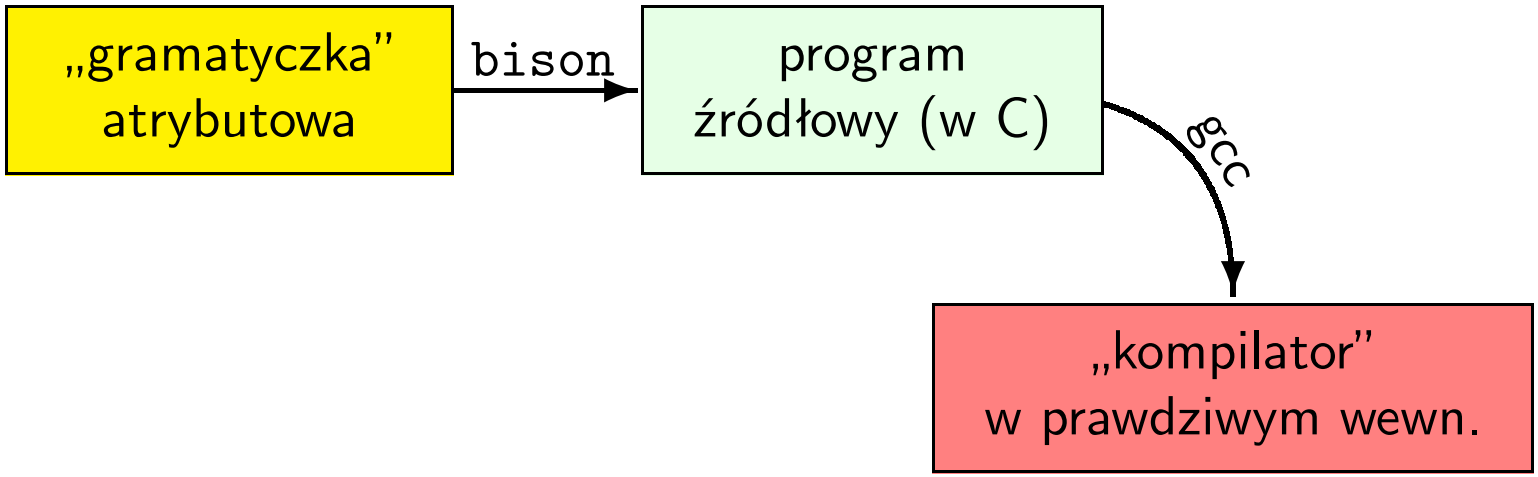
Działanie BISONa — kompilacja

„gramatyczka” atrybutowa

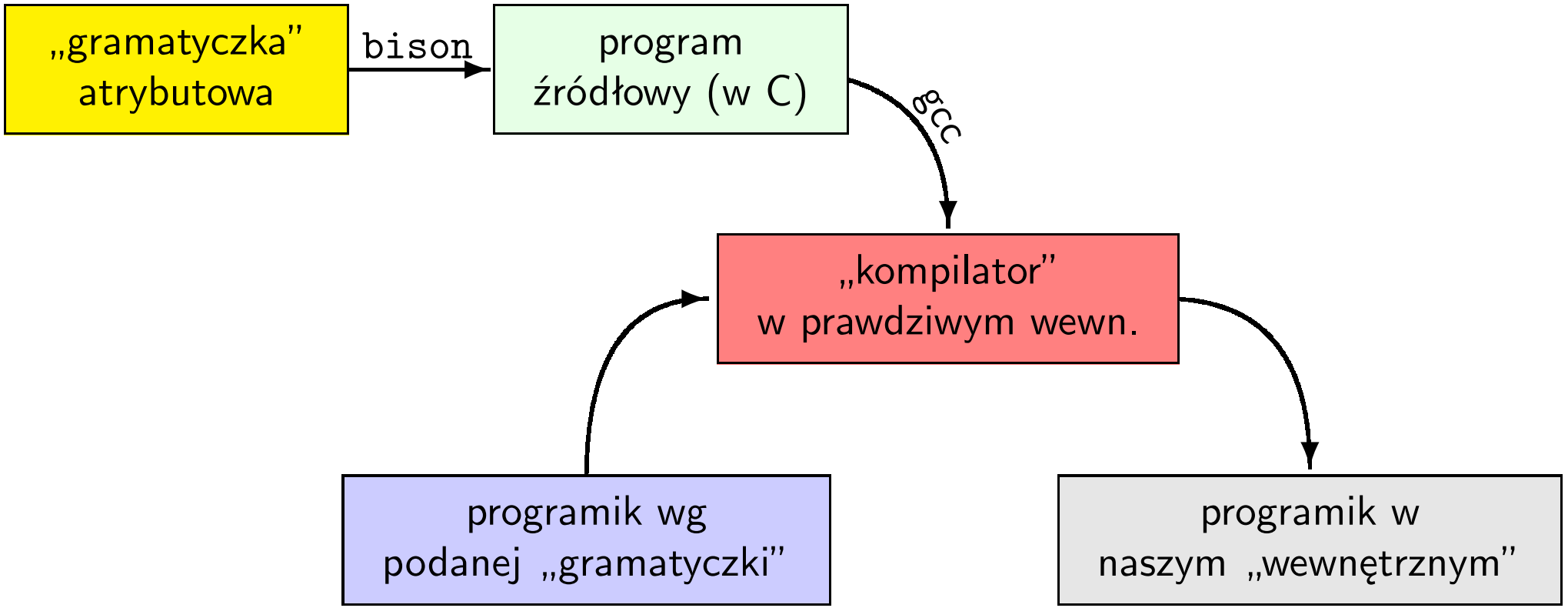
Działanie BISONa — kompilacja



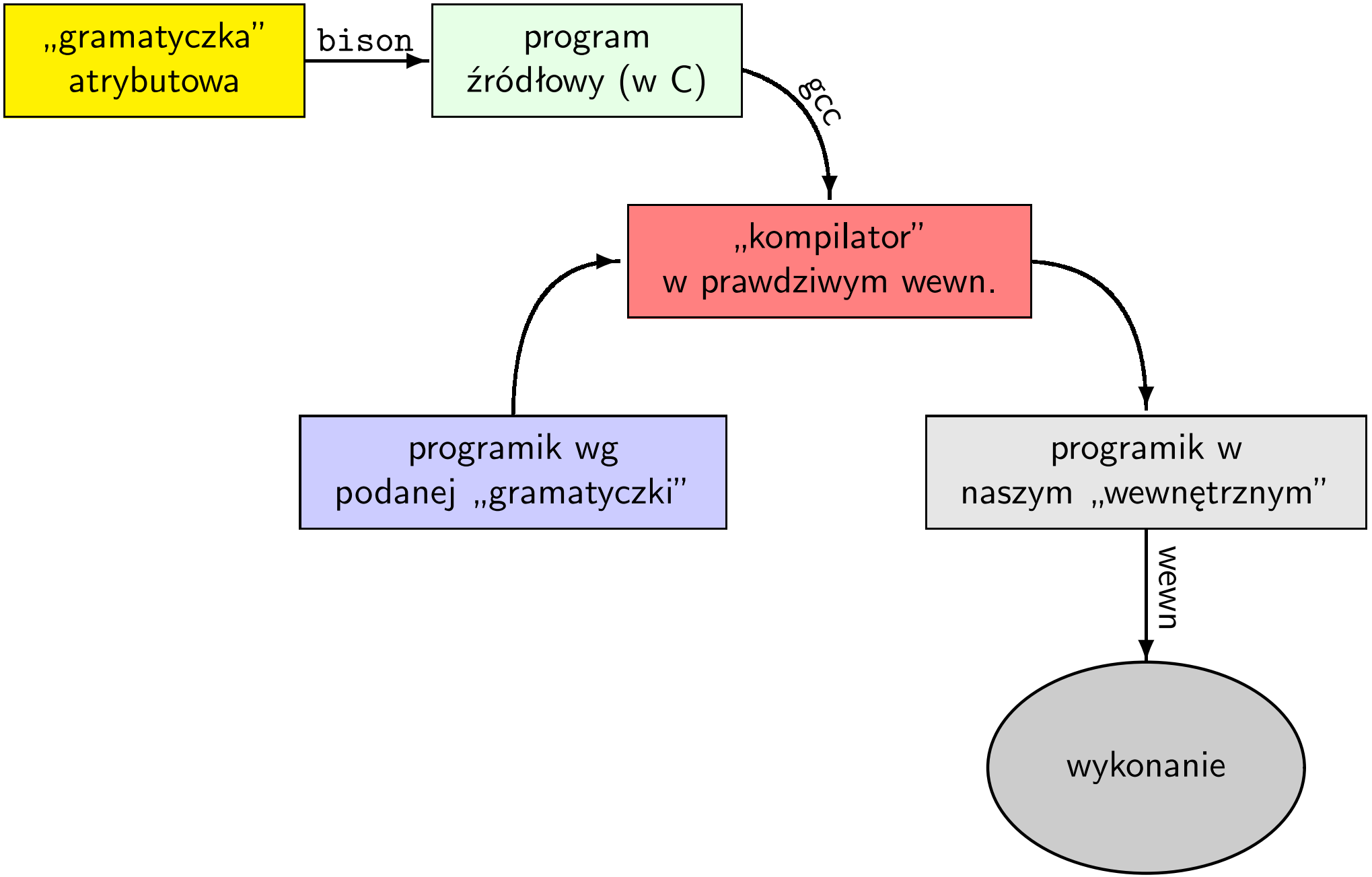
Działanie BISONa — kompilacja



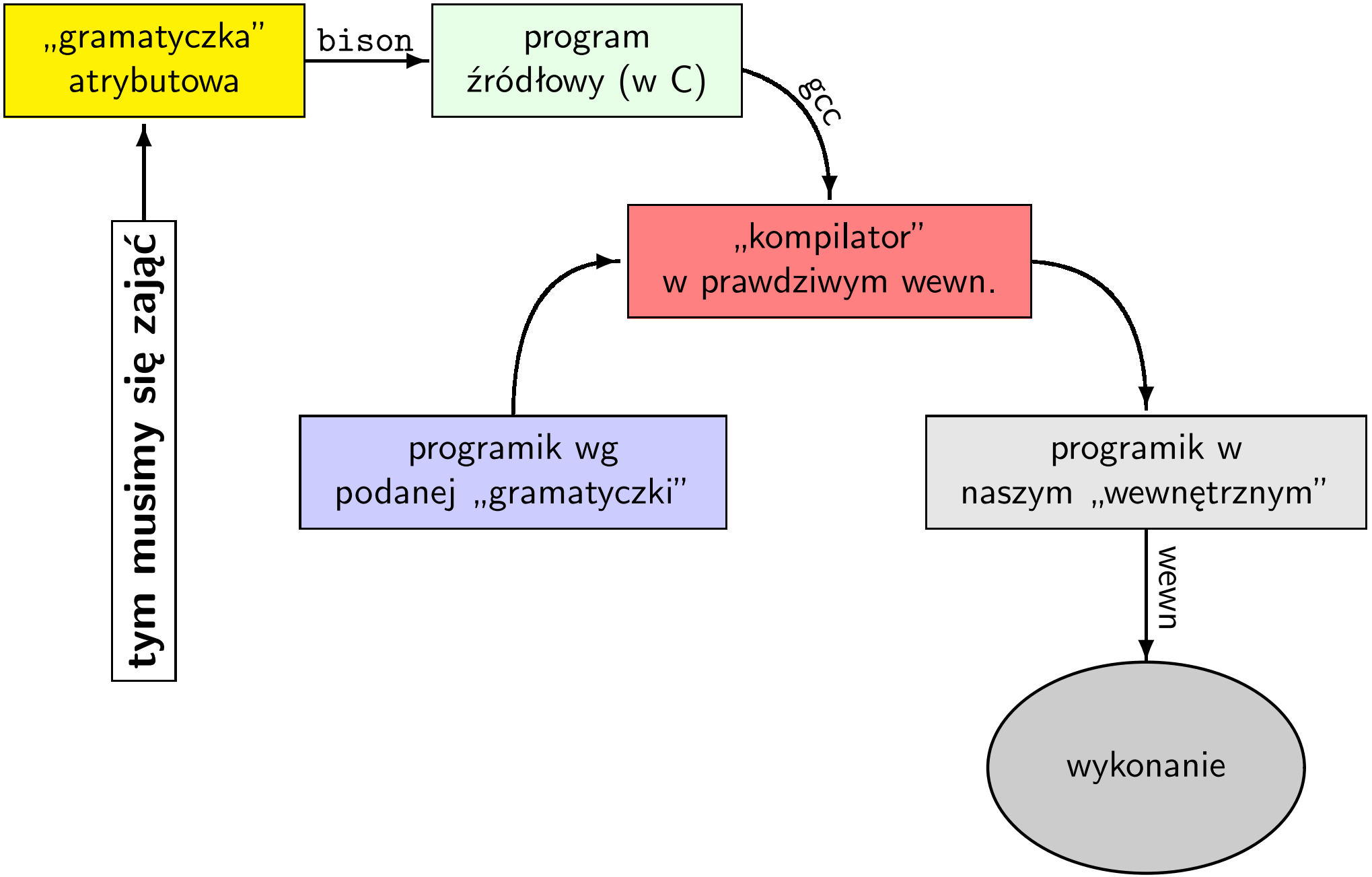
Działanie BISONa — kompilacja



Działanie BISONa — kompilacja



Działanie BISONa — kompilacja



Działanie BISONa — kompilacja

Atrybut:

- **dla cyfry i liczby:** — wartość

Działanie BISONa — kompilacja

Atrybut:

- **dla cyfry i liczby:** — wartość;
- **dla zmiennej:** — adres zmiennej

Działanie BISONa — kompilacja

Atrybut:

- **dla cyfry i liczby:** — wartość;
- **dla zmiennej:** — adres zmiennej;
- **dla komendy:** — liczba komórek przekładu.

Działanie BISONa — kompilacja

Atrybut:

- **dla cyfry i liczby:** — wartość;
- **dla zmiennej:** — adres zmiennej;
- **dla komendy:** — liczba komórek przekładu.

Przykłady:

```
komenda: zmienna '=' zmienna ';'
```

Działanie BISONa — kompilacja

Atrybut:

- **dla cyfry i liczby:** — wartość;
- **dla zmiennej:** — adres zmiennej;
- **dla komendy:** — liczba komórek przekładu.

Przykłady:

```
komenda: zmienna '=' zmienna ';'        {napisać „loada " , $3
                                         napisać „store " , $1
                                         $$ = 2; }
```

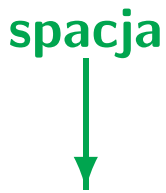
Działanie BISONa — kompilacja

Atrybut:

- **dla cyfry i liczby:** — wartość;
- **dla zmiennej:** — adres zmiennej;
- **dla komendy:** — liczba komórek przekładu.

Przykłady:

```
komenda: zmienna '=' zmienna ';'
```



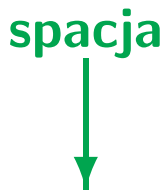

```
{napisać „loada ” , $3
 napisać „store ” , $1
 $$ = 2; }
```

Działanie BISONa — kompilacja

Atrybut:

- **dla cyfry i liczby:** — wartość;
- **dla zmiennej:** — adres zmiennej;
- **dla komendy:** — liczba komórek przekładu.

Przykłady:



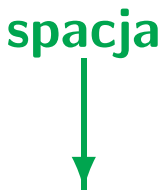

```
komenda: zmienna '=' zmienna ';'          {napisać „loada " , $3
                                           napisać „store " , $1
                                           $$ = 2; }

       | zmienna '=' zmienna '+' zmienna ';'
```

Działanie BISONa — kompilacja

Atrybut:

- **dla cyfry i liczby:** — wartość;
- **dla zmiennej:** — adres zmiennej;
- **dla komendy:** — liczba komórek przekładu.

Przykłady:



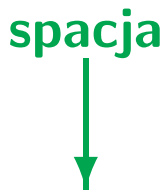

```
komenda: zmienna '=' zmienna ';'          {napisać „loada " , $3
                                           napisać „store " , $1
                                           $$ = 2; }

       | zmienna '=' zmienna '+' zmienna ';'  {napisać „loada " , $3
                                               napisać „add "   , $5
                                               napisać „store " , $1
                                               $$ = 3; }
```

Działanie BISONa — kompilacja

Atrybut:

- **dla cyfry i liczby:** — wartość;
- **dla zmiennej:** — adres zmiennej;
- **dla komendy:** — liczba komórek przekładu.

Przykłady:



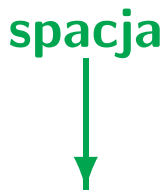

```
komenda: zmienna '=' zmienna ';'                {napisać „loada " , $3
                                                 napisać „store " , $1
                                                 $$ = 2; }

       | zmienna '=' zmienna '+' zmienna ';'     {napisać „loada " , $3
                                                 napisać „add "   , $5
                                                 napisać „store " , $1
                                                 $$ = 3; }

       | zmienna '=' zmienna '+' liczba ';'
```

Działanie BISONa — kompilacja

Atrybut:

- **dla cyfry i liczby:** — wartość;
- **dla zmiennej:** — adres zmiennej;
- **dla komendy:** — liczba komórek przekładu.

Przykłady:



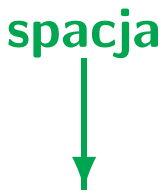

```
komenda: zmienna '=' zmienna ';'                {napisać „loada " , $3
                                                 napisać „store " , $1
                                                 $$ = 2; }

       | zmienna '=' zmienna '+' zmienna ';'     {napisać „loada " , $3
                                                 napisać „add "   , $5
                                                 napisać „store " , $1
                                                 $$ = 3; }

       | zmienna '=' zmienna '+' liczba ';'      {napisać „loadn " , $5
                                                 napisać „add "   , $3
                                                 napisać „store " , $1
                                                 $$ = 3; }
```

Działanie BISONa — kompilacja

Atrybut:

- **dla cyfry i liczby:** — wartość;
- **dla zmiennej:** — adres zmiennej;
- **dla komendy:** — liczba komórek przekładu.

Przykłady:




```
komenda: zmienna '=' zmienna ';'                {napisać „loada " , $3
                                                 napisać „store " , $1
                                                 $$ = 2; }

       | zmienna '=' zmienna '+' zmienna ';'     {napisać „loada " , $3
                                                 napisać „add "   , $5
                                                 napisać „store " , $1
                                                 $$ = 3; }

       | zmienna '=' zmienna '+' liczba ';'      {napisać „loadn " , $5
                                                 napisać „add "   , $3
                                                 napisać „store " , $1
                                                 $$ = 3; }
```

Analiza leksykalna

Analiza leksykalna

Dotąd prymitywna:

```
#define YYSTYPE int
. . . . .
YYSTYPE yylex (void) {
  int zn;
  do  { zn = getchar(); }  while (isspace(zn));
  if (zn == EOF) return 0;
  else return zn;
}
```

stanowiąca część parsera.

Analiza leksykalna

Dotąd prymitywna:

```
#define YYSTYPE int
. . . . .
YYSTYPE yylex (void) {
  int zn;
  do  { zn = getchar(); }  while (isspace(zn));
  if (zn == EOF) return 0;
  else return zn;
}
```

stanowiąca część parsera.

Można (warto!) wyodrębnić ją w osobny moduł.

— np. po to, żeby leksemy mogły być wieloznakowe...

Analiza leksykalna

Analiza leksykalna

Fast

L

E

X

Analiza leksykalna

Fast Prekursor: **LEX**

L

E

X

Analiza leksykalna

Fast
L
E
X

Prekursor: **LEX**

Wersja GNU:

FLEX — `www.cs.princeton.edu/~appel/modern/c/software/flex`

Analiza leksykalna

Fast
L
E
X

Prekursor: **LEX**

Wersja GNU:

FLEX — `www.cs.princeton.edu/~appel/modern/c/software/flex`

— wyszukuje wzorce regularne i wykonuje akcje; w najprostszym przypadku zastępuje

Analiza leksykalna

Fast
L
E
X

Prekursor: **LEX**

Wersja GNU:
FLEX — `www.cs.princeton.edu/~appel/modern/c/software/flex`

— wyszukuje wzorce regularne i wykonuje akcje; w najprostszym przypadku zastępuje

---

```
definicje
%%
reguły
%%
kod użytkownika
```

Analiza leksykalna

Fast
L
E
X

Prekursor: **LEX**

Wersja GNU:
FLEX — `www.cs.princeton.edu/~appel/modern/c/software/flex`

— wyszukuje wzorce regularne i wykonuje akcje; w najprostszym przypadku zastępuje

---

definicje
%%
reguły
%%
kod użytkownika

Najważniejsza część: *reguły*

Analiza leksykalna

Fast
L
E
X

Prekursor: **LEX**

Wersja GNU:
FLEX — `www.cs.princeton.edu/~appel/modern/c/software/flex`

— wyszukuje wzorce regularne i wykonuje akcje; w najprostszym przypadku zastępuje

---

definicje
%%
reguły
%%
kod użytkownika

Najważniejsza część: *reguły* składające się z

- *wzorca* — jego napotkanie w tekście wejściowym ma spowodować jakąś akcję

Analiza leksykalna

Fast
L
E
X

Prekursor: **LEX**

Wersja GNU:
FLEX — `www.cs.princeton.edu/~appel/modern/c/software/flex`

— wyszukuje wzorce regularne i wykonuje akcje; w najprostszym przypadku zastępuje

---

```
definicje
%%
reguły
%%
kod użytkownika
```

Najważniejsza część: *reguły* składające się z

- *wzorca* — jego napotkanie w tekście wejściowym ma spowodować jakąś akcję;
- *akcji* — czynności, które należy podjąć po rozpoznaniu wzorca w tekście wejściowym (opisane w C).

Analiza leksykalna

Przykład:

1. do pliku `test.l` to

```
[0-9]+      printf("LICZBA");
[+\-]       printf("OP");
[A-Z][a-z]+ printf("KTOŚ");
```

Analiza leksykalna

Przykład:

1. do pliku `test.l` to
2. komenda `flex test.l` produkuje program `lex.yy.c`

```
[0-9]+      printf("LICZBA");
[+\-]       printf("OP");
[A-Z][a-z]+ printf("KTOŚ");
```

Analiza leksykalna

Przykład:

1. do pliku `test.l` to 
2. komenda `flex test.l` produkuje program `lex.yy.c`
3. komenda `gcc lex.yy.c -lfl` produkuje przekład `a.out`

```
[0-9]+      printf("LICZBA");
[+\-]       printf("OP");
[A-Z][a-z]+ printf("KTOŚ");
```

Analiza leksykalna

Przykład:

1. do pliku `test.l` to
2. komenda `flex test.l` produkuje program `lex.yy.c`
3. komenda `gcc lex.yy.c -lfl` produkuje przekład `a.out`

```
[0-9]+      printf("LICZBA");
[+\-]       printf("OP");
[A-Z][a-z]+ printf("KTOŚ");
```

Teraz komenda `./a.out < dane` przekształca np.

```
kto wie, ile jest
  (15 - 8) + 6 ?
Jasio wie, Szymek nie wie.
```

dane

Analiza leksykalna

Przykład:

1. do pliku `test.l` to
2. komenda `flex test.l` produkuje program `lex.yy.c`
3. komenda `gcc lex.yy.c -lfl` produkuje przekład `a.out`

```
[0-9]+        printf("LICZBA");
[+\-]         printf("OP");
[A-Z][a-z]+   printf("KTOŚ");
```

Teraz komenda `./a.out < dane` przekształca np.

```
kto wie, ile jest
  (15 - 8) + 6 ?
Jasio wie, Szymek nie wie.
```

dane

w

```
kto wie, ile jest
  (LICZBA OP LICZBA) OP LICZBA ?
KTOŚ wie, KTOŚ nie wie.
```

wynik

Analiza leksykalna

Przykład:

1. do pliku `test.l` to
2. komenda `flex test.l` produkuje program `lex.yy.c`
3. komenda `gcc lex.yy.c -lfl` produkuje przekład `a.out`

```
[0-9]+        printf("LICZBA");
[+\-]         printf("OP");
[A-Z][a-z]+   printf("KTOŚ");
```

Teraz komenda `./a.out < dane` przekształca np.

```
kto wie, ile jest
  (15 - 8) + 6 ?
Jasio wie, Szymek nie wie.
```

dane

w

```
kto wie, ile jest
  (LICZBA OP LICZBA) OP LICZBA ?
KTOŚ wie, KTOŚ nie wie.
```

wynik

Działanie FLEXa

- Akcje FLEXa *mogą* (ale nie muszą) polegać na zastępowaniu fragmentów tekstu innymi fragmentami. Mogą to być dowolne komendy języka C.

Działanie FLEXa

- Akcje FLEXa *mogą* (ale nie muszą) polegać na zastępowaniu fragmentów tekstu innymi fragmentami. Mogą to być dowolne komendy języka C.
- Wzorce muszą być *wyrażeniami regularnymi*. FLEX rozumie wiele rozmaitych rodzajów wyrażeń regularnych.

Działanie FLEXa

- Akcje FLEXa *mogą* (ale nie muszą) polegać na zastępowaniu fragmentów tekstu innymi fragmentami. Mogą to być dowolne komendy języka C.
- Wzorce muszą być *wyrażeniami regularnymi*. FLEX rozumie wiele rozmaitych rodzajów wyrażeń regularnych.
- Część definicyjna (przed regułami) może służyć na przykład do tego, żeby podefiniować typy i funkcje potrzebne w akcjach reguł, podeklarować zmienne pomocnicze i wykonać inicjalizacje.

Działanie FLEXa

- Akcje FLEXa *mogą* (ale nie muszą) polegać na zastępowaniu fragmentów tekstu innymi fragmentami. Mogą to być dowolne komendy języka C.
- Wzorce muszą być *wyrażeniami regularnymi*. FLEX rozumie wiele rozmaitych rodzajów wyrażeń regularnych.
- Część definicyjna (przed regułami) może służyć na przykład do tego, żeby podefiniować typy i funkcje potrzebne w akcjach reguł, podeklarować zmienne pomocnicze i wykonać inicjalizacje.
- Część kodu użytkownika (za regułami) może służyć na przykład do tego, żeby porządnie zakończyć działanie analizatora.

Działanie FLEXa

- Akcje FLEXa *mogą* (ale nie muszą) polegać na zastępowaniu fragmentów tekstu innymi fragmentami. Mogą to być dowolne komendy języka C.
- Wzorce muszą być *wyrażeniami regularnymi*. FLEX rozumie wiele rozmaitych rodzajów wyrażeń regularnych.
- Część definicyjna (przed regułami) może służyć na przykład do tego, żeby podefiniować typy i funkcje potrzebne w akcjach reguł, podeklarować zmienne pomocnicze i wykonać inicjalizacje.
- Część kodu użytkownika (za regułami) może służyć na przykład do tego, żeby porządnie zakończyć działanie analizatora.
- Kod w języku C dostarczony przez użytkownika jest przez FLEXa umieszczany bez zmian w gotowym analizatorze `lex.yy.c` . Jeśli są w nim błędy, to zostaną one zasygnalizowane dopiero w czasie kompilacji translatorem `gcc` .

BISON z FLEXem

BISON z FLEXem

Translator utworzony przez Bisona czyta tekst wejściowy (w języku źródłowym) leksem za leksemem

BISON z FLEXem

Translator utworzony przez Bisona czyta tekst wejściowy (w języku źródłowym) leksem za leksemem — każde wywołanie funkcji

```
YYSTYPE yylex (void)
```

(typ `YYSTYPE` sami definiujemy) dostarcza mu jeden leksem.

BISON z FLEXem

Translator utworzony przez Bisona czyta tekst wejściowy (w języku źródłowym) leksem za leksemem — każde wywołanie funkcji

```
YYSTYPE yylex (void)
```

(typ `YYSTYPE` sami definiujemy) dostarcza mu jeden leksem.

Jeśli mamy leksemy więcej niż 1-znakowe, to najwygodniej zdefiniować tą funkcję przez Flexa.

BISON z FLEXem

- w danych do Bisona *plik*`.y` deklarujemy każdy taki leksem:
  `%token PETLA`

BISON z FLEXem

- w danych do Bisona *plik*`.y` deklarujemy każdy taki leksem:
  `%token PETLA`
- tam, gdzie leksem ma wystąpić w gramatyce, piszemy tą zadeklarowaną nazwę:
  `PETLA '(' zmienna ')' '{' komendy '}'`

BISON z FLEXem

- w danych do Bisona *plik*`.y` deklarujemy każdy taki leksem:
  `%token PETLA`
- tam, gdzie leksem ma wystąpić w gramatyce, piszemy tą zadeklarowaną nazwę:
  `PETLA '(' zmienna ')' '{' komendy '}'`
- przetwarzamy dane do Bisona komendą:
  `bison -d` *plik.y*
  — to tworzy program w C *plik*`.tab.c` oraz plik nagłówkowy *plik*`.tab.h`

BISON z FLEXem

- w danych do Bisona *plik*`.y` deklarujemy każdy taki leksem:
  `%token PETLA`
- tam, gdzie leksem ma wystąpić w gramatyce, piszemy tą zadeklarowaną nazwę:
  `PETLA '(' zmienna ')' '{' komendy '}'`
- przetwarzamy dane do Bisona komendą:
  `bison -d` *plik.y*
  — to tworzy program w C *plik*`.tab.c` oraz plik nagłówkowy *plik*`.tab.h`
- we wstępnej części danych do Flexa *plik*`.l` odwołujemy się do pliku nagłówkowego:
  `#include "`*plik*`.tab.h"`

BISON z FLEXem

- w danych do Bisona *plik*`.y` deklarujemy każdy taki leksem:
  `%token PETLA`
- tam, gdzie leksem ma wystąpić w gramatyce, piszemy tą zadeklarowaną nazwę:
  `PETLA '(' zmienna ')' '{' komendy '}'`
- przetwarzamy dane do Bisona komendą:
  `bison -d` *plik.y*
  — to tworzy program w C *plik*`.tab.c` oraz plik nagłówkowy *plik*`.tab.h`
- we wstępnej części danych do Flexa *plik*`.l` odwołujemy się do pliku nagłówkowego:
  `#include "`*plik*`.tab.h"`
  podajemy reguły rozpoznawania wieloznakowych leksemów:
  `"WHILE" { return PETLA; }`

BISON z FLEXem

- w danych do Bisona *plik*`.y` deklarujemy każdy taki leksem:
  `%token PETLA`
- tam, gdzie leksem ma wystąpić w gramatyce, piszemy tą zadeklarowaną nazwę:
  `PETLA '(' zmienna ')' '{' komendy '}'`
- przetwarzamy dane do Bisona komendą:
  `bison -d` *plik.y*
  — to tworzy program w C *plik*`.tab.c` oraz plik nagłówkowy *plik*`.tab.h`
- we wstępnej części danych do Flexa *plik*`.l` odwołujemy się do pliku nagłówkowego:
  `#include "`*plik*`.tab.h"`
  podajemy reguły rozpoznawania wieloznakowych leksemów:
  `"WHILE" { return PETLA; }`
  i **pomijamy** `main` dla tego pliku

BISON z FLEXem

- w danych do Bisona *plik*`.y` deklarujemy każdy taki leksem:
  `%token PETLA`
- tam, gdzie leksem ma wystąpić w gramatyce, piszemy tą zadeklarowaną nazwę:
  `PETLA '(' zmienna ')' '{' komendy '}'`
- przetwarzamy dane do Bisona komendą:
  `bison -d` *plik.y*
  — to tworzy program w C *plik*`.tab.c` oraz plik nagłówkowy *plik*`.tab.h`
- we wstępnej części danych do Flexa *plik*`.l` odwołujemy się do pliku nagłówkowego:
  `#include "`*plik*`.tab.h"`
  podajemy reguły rozpoznawania wieloznakowych leksemów:
  `"WHILE" { return PETLA; }`
  i **pomijamy** `main` dla tego pliku
- przetwarzamy dane do Flexa komendą:
  `flex` *plik*`.l`
  — to tworzy program w C `lex.yy.c`

BISON z FLEXem

- w danych do Bisona *plik*`.y` deklarujemy każdy taki leksem:
  `%token PETLA`
- tam, gdzie leksem ma wystąpić w gramatyce, piszemy tą zadeklarowaną nazwę:
  `PETLA '(' zmienna ')' '{' komendy '}'`
- przetwarzamy dane do Bisona komendą:
  `bison -d` *plik.y*
  — to tworzy program w C *plik*`.tab.c` oraz plik nagłówkowy *plik*`.tab.h`
- we wstępnej części danych do Flexa *plik*`.l` odwołujemy się do pliku nagłówkowego:
  `#include "`*plik*`.tab.h"`
  podajemy reguły rozpoznawania wieloznakowych leksemów:
  `"WHILE" { return PETLA; }`
  i **pomijamy** `main` dla tego pliku
- przetwarzamy dane do Flexa komendą:
  `flex` *plik*`.l`
  — to tworzy program w C `lex.yy.c`
- oba programy w C kompilujemy łącznie, razem z opcją biblioteki Flexa:
  `gcc` *plik*`.tab.c lex.yy.c -lfl`
  — to tworzy program w C `a.out` , stanowiący żądany kompilator

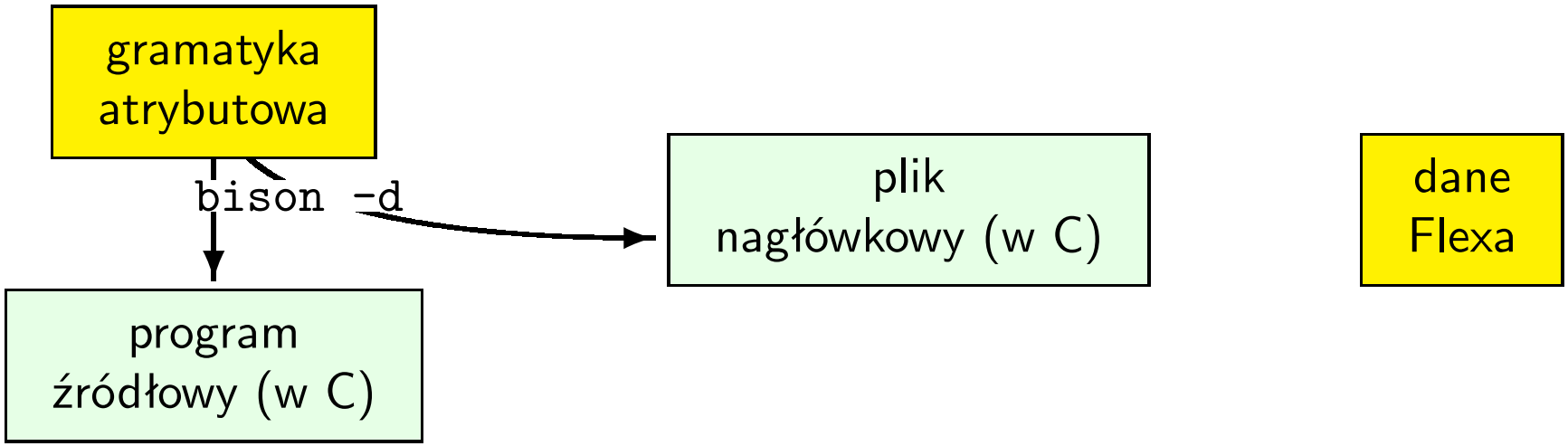
BISON z FLEXem

BISON z FLEXem

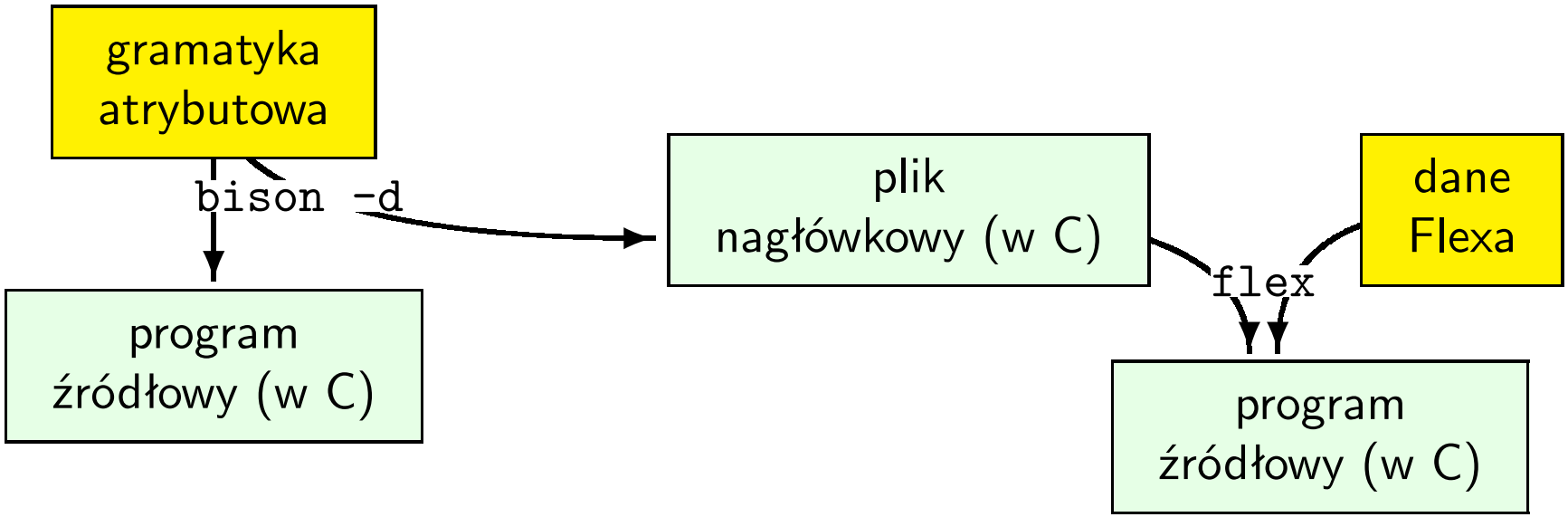
gramatyka atrybutowa

dane Flexa

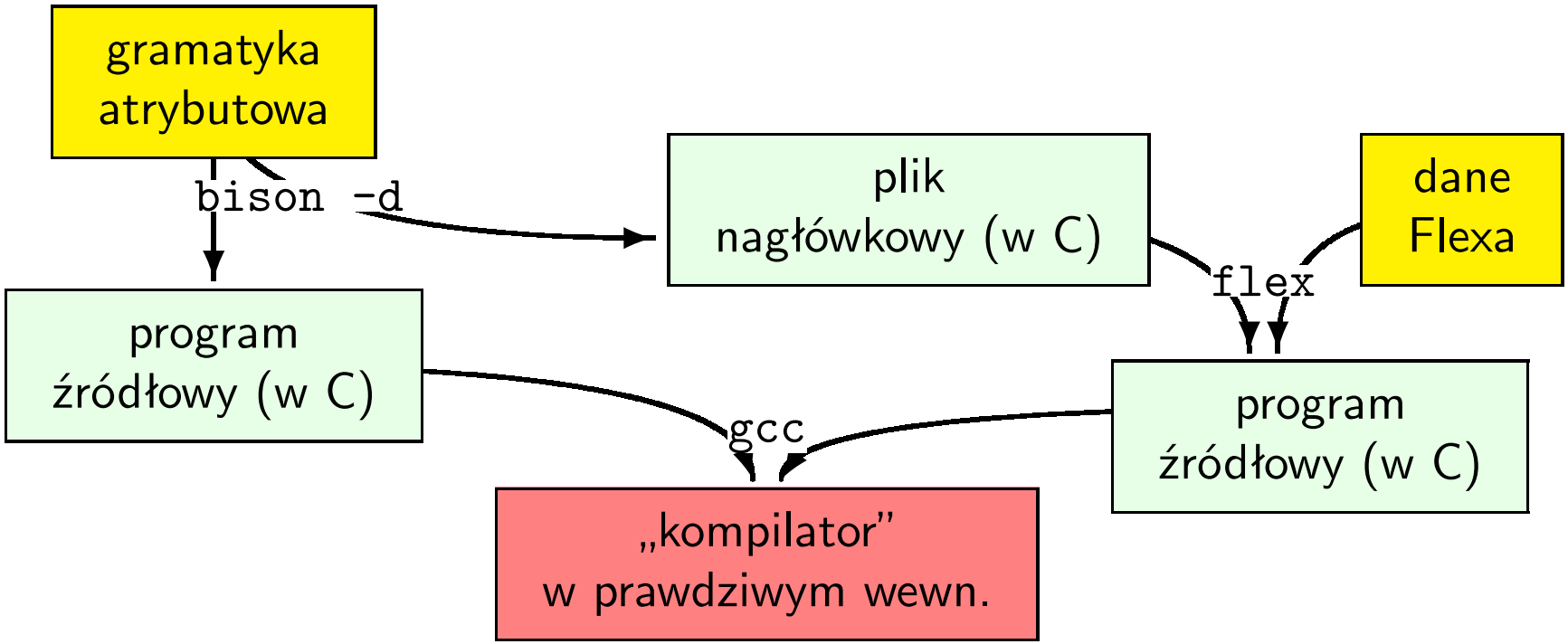
BISON z FLEXem



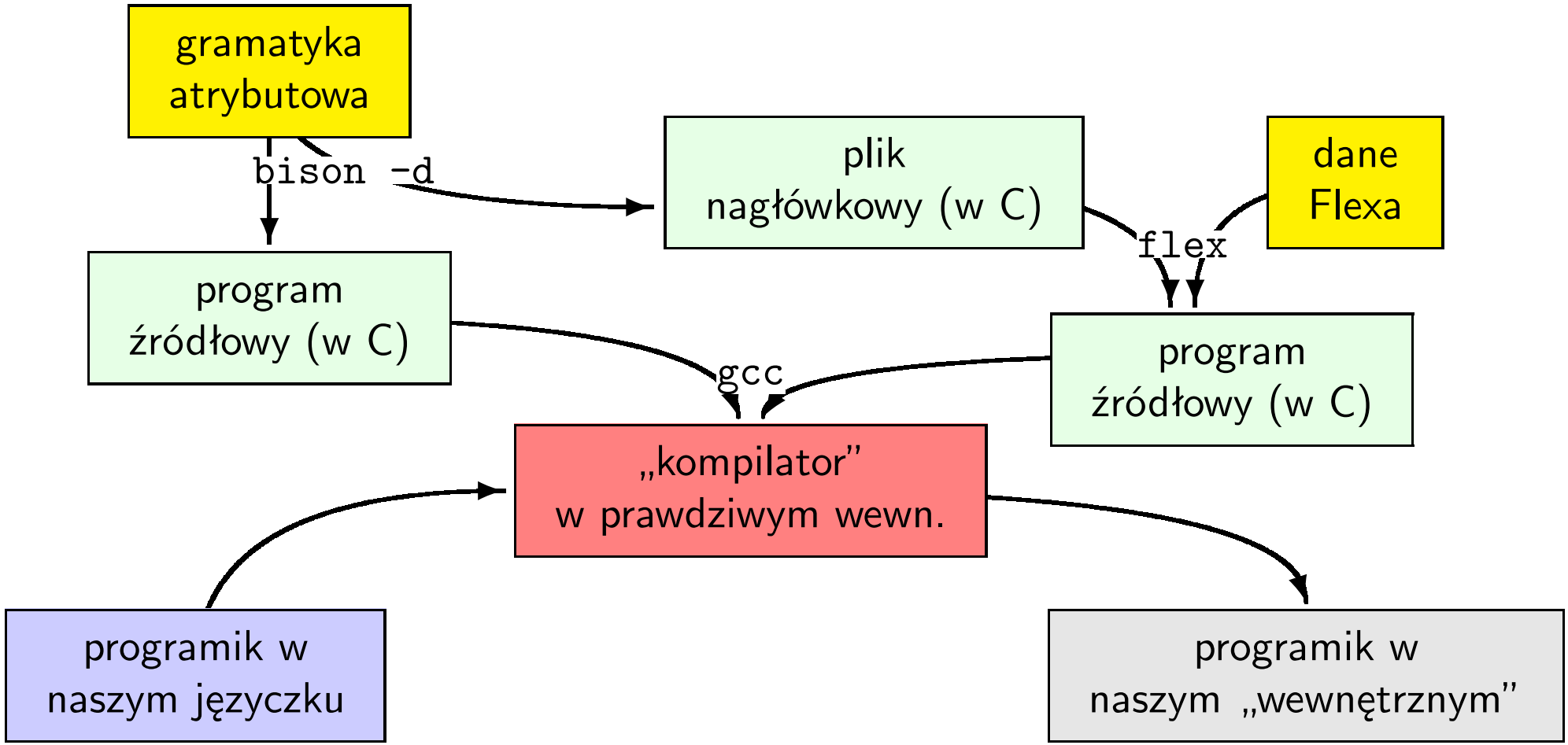
BISON z FLEXem



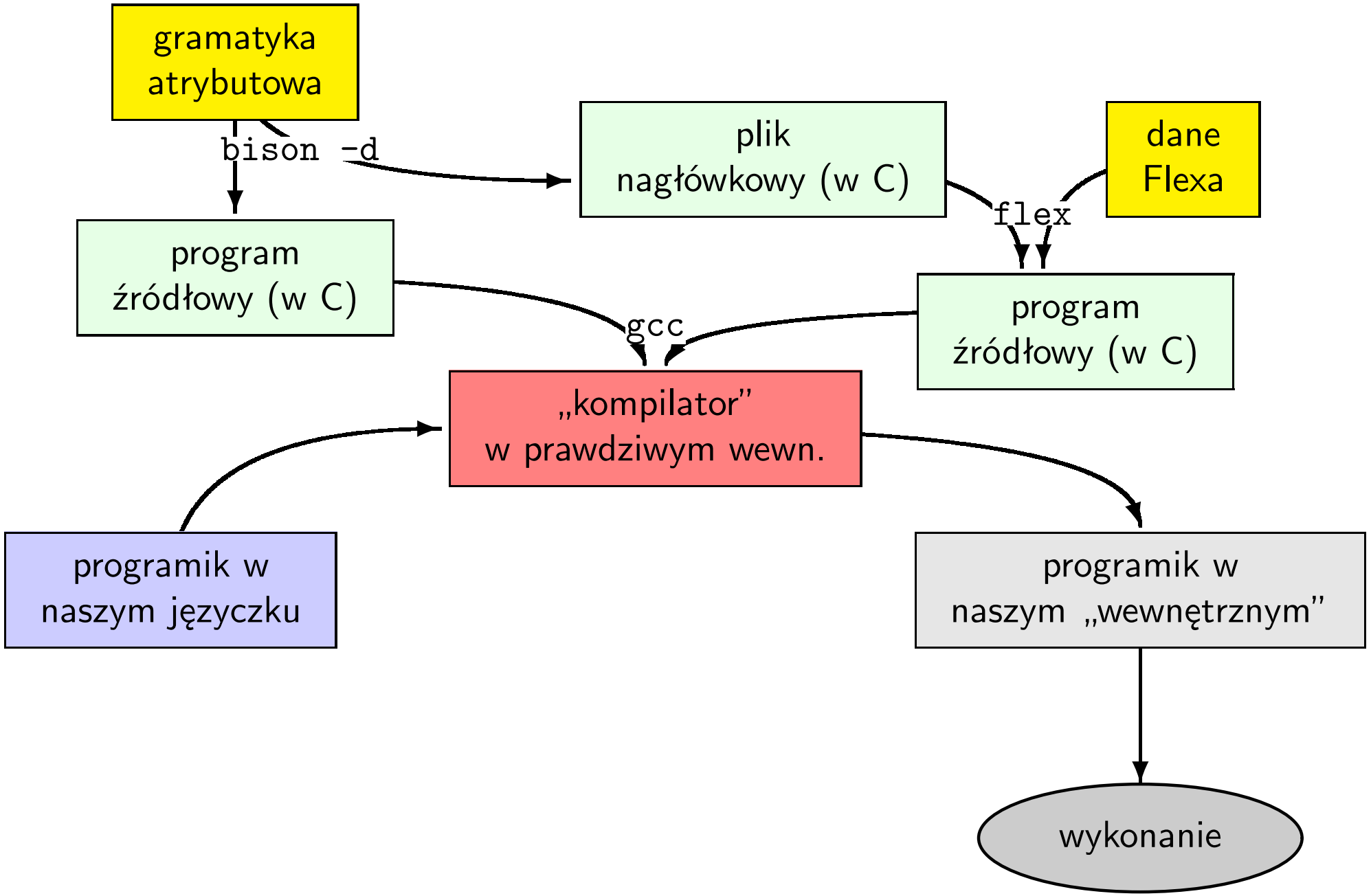
BISON z FLEXem



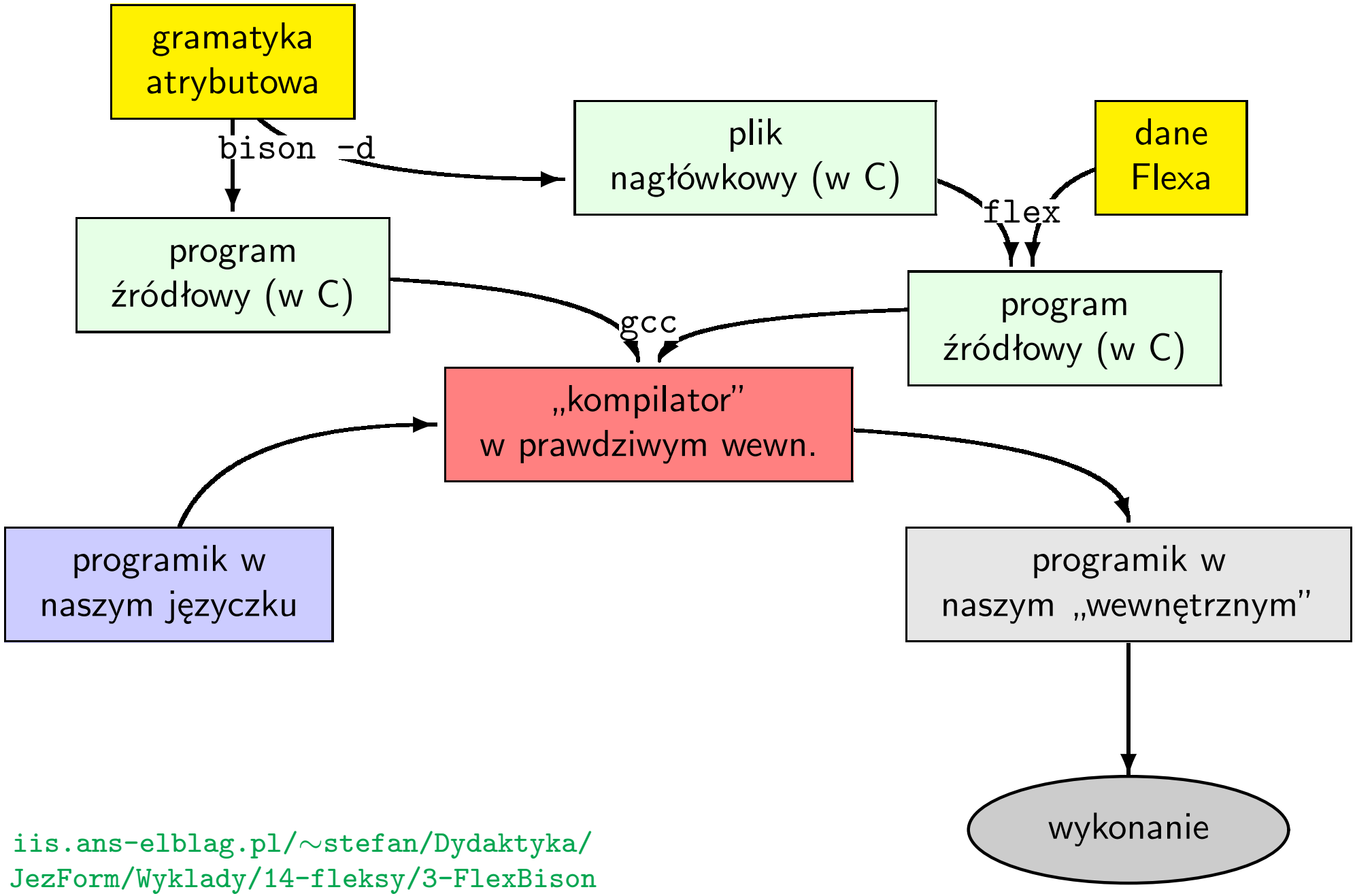
BISON z FLEXem



BISON z FLEXem



BISON z FLEXem



iis.ans-elblag.pl/∼stefan/Dydaktyka/JezForm/Wyklady/14-fleksy/3-FlexBison