# Translacja sterowana składnią w metodzie zstępującej

Wojciech Complak
Wojciech.Complak@cs.put.poznan.pl

UCZELNIA
ONLINE

## Plan wykładu

- translacja sterowana składnią
- definicje sterowane składnią i schematy translacji
- atrybuty syntetyzowane i dziedziczone
- definicje S-atrybutowe i L-atrybutowe
- implementacja translacji sterowanej składnią w metodzie zstępującej w języku C i za pomocą generatora LLgen



W ramach wykładu zostaną omówione następujące zagadnienia:

•czym jest translacja sterowana składnią ?

•definicje sterowane składnią i schematy translacji

•atrybuty syntetyzowane i dziedziczone

•definicje S-atrybutowe i L-atrybutowe

•implementacja translacji sterowanej składnią w metodzie zstępującej w języku C i za pomocą generatora LLgen

## Translacja sterowana składnią

- translacja sterowana składnią to translacja języków oparta o gramatyki bezkontekstowe, w której:
  - z konstrukcjami języka programowania wiązana jest pewna informacja poprzez dołączenie atrybutów do symboli gramatyki reprezentujących te konstrukcje
  - wartości atrybutów obliczane są przez tzw. reguły semantyczne związane z produkcjami gramatyki

Translacja sterowana składnią w metodzie zstępującej (3)

Translacja sterowana składnią to translacja języków oparta o gramatyki bezkontekstowe, w której:

•z konstrukcjami języka programowania wiązana jest pewna informacja poprzez dołączenie atrybutów do symboli gramatyki reprezentujących te konstrukcje

•wartości atrybutów obliczane są przez tzw. reguły semantyczne związane z produkcjami gramatyki

## Definicje sterowane składnią i schematy translacji

- istnieją dwie notacje łączące reguły semantyczne z produkcjami
  - definicje sterowane składnią – ukrywają wiele szczegółów implementacyjnych, nie wymagają jawnego określania kolejność obliczania reguł semantycznych
  - schematy translacji – wskazują kolejność ewaluacji reguł semantycznych dzięki czemu widocznych jest więcej szczegółów implementacyjnych



Istnieją dwie notacje służące do wiązania reguł semantycznych z produkcjami:

•definicje sterowane składnią – notacja wysokopoziomowa ukrywająca wiele szczegółów implementacyjnych i nie wymagająca jawnego określania kolejności obliczania reguł semantycznych

•schematy translacji – notacja niskopoziomowa wskazująca kolejność ewaluacji reguł semantycznych dzięki czemu widocznych jest więcej szczegółów implementacyjnych

## Definicje sterowane składnią

- definicje sterowane składnią są uogólnieniem gramatyki bezkontekstowej, w której z każdym symbolem związany jest pewien zbiór atrybutów
- atrybuty dzielimy na syntetyzowane i dziedziczone
- atrybuty mogą reprezentować dowolne wielkości (napisy, liczby, typy, adresy …)
- wartości atrybutów w węźle drzewa wywodu są określane przez reguły semantyczne związane z produkcją przypisaną do tego węzła



Definicje sterowane składnią są uogólnieniem gramatyki bezkontekstowej, w której z każdym symbolem gramatyki związany jest pewien zbiór atrybutów.

Atrybuty – w zależności od sposobu ich ewaluacji – dzielimy na syntetyzowane i dziedziczone.

Atrybutów możemy użyć do przechowywania informacji dowolnego rodzaju, np. napisów, liczb, typów, adresów …

Wartości atrybutów w węźle drzewa wywodu są określane przez reguły semantyczne związane z produkcją przypisaną do tego węzła drzewa.

## Atrybuty syntetyzowane i dziedziczone

- w definicji sterowanej składnią z każdą produkcją gramatyki $B \rightarrow X_1 X_2 \ldots X_n$ jest związany zbiór reguł semantycznych o postaci:
  $b := f(p_1, p_2, \ldots, p_n)$, gdzie:
  - b jest atrybutem syntetyzowanym symbolu B, a $p_1, p_2, \ldots, p_n$ są atrybutami symboli $X_1, X_2, \ldots, X_n$
  - b jest atrybutem dziedziczonym symbolu $X_i$, a $p_1, p_2, \ldots, p_n$ są atrybutami symboli B, $X_1, X_2, \ldots, X_n$

Translacja sterowana składnią w metodzie zstępującej (6)

W definicji sterowanej składnią z każdą produkcją gramatyki możemy związać zbiór reguł semantycznych (akcji), w których obliczane są wartości atrybutów.

Wartości atrybutów syntetyzowanych obliczane są na podstawie wartości atrybutów dzieci tego węzła w drzewie wywodu. Na poziomie reguły semantycznej oznacza to, że atrybut symbolu stojącego po **lewej** stronie produkcji jest funkcją atrybutów symboli stojących po prawej stronie produkcji.

Wartości atrybutów dziedziczonych obliczane są na podstawie wartości atrybutów sąsiadów i rodzica tego węzła w drzewie wywodu. Na poziomie reguły semantycznej oznacza to, że atrybut symbolu stojącego po **prawej** stronie produkcji jest funkcją atrybutów symbolu po lewej stronie produkcji i atrybutów symboli po prawej stronie produkcji.

Przyjmuje się, że terminale nie mogą mieć atrybutów dziedziczonych (w definicji sterowanej składnią nie ma reguł semantycznych dla terminali). Zwykle zakłada się również, że aksjomat gramatyki nie ma atrybutów dziedziczonych.

## Definicje S-atrybutowe i L-atrybutowe

- wśród definicji sterowanych składnią wyróżniamy dwie podklasy:
  - S-atrybutowe – tylko atrybuty syntetyzowane
  - L-atrybutowe – każdy atrybut może być:
    - atrybutem syntetyzowanym albo
    - atrybutem dziedziczonym symbolu $X_i$ w produkcji $B \rightarrow X_1 X_2 \ldots X_i \ldots X_n$, który zależy od atrybutów symboli $X_1 X_2 \ldots X_{i-1}$ oraz atrybutu dziedziczonego symbolu B



Wśród definicji sterowanych składnią wyróżniamy dwie podklasy:

•definicje S-atrybutowe, w których używane są jedynie atrybuty syntetyzowane

•definicje L-atrybutowe, w których każdy atrybut może być:
atrybutem syntetyzowanym albo
atrybutem dziedziczonym symbolu stojącego po prawej stronie produkcji, który zależy od atrybutów symboli stojących po prawej stronie produkcji na lewo od niego i od atrybutu dziedziczonego symbolu stojącego po lewej stronie produkcji.

Każda definicja S-atrybutowa jest również L-atrybutowa

Definicja L-atrybutowa jest szczególnie istotna z praktycznego punktu widzenia – translacja może być wykonywana w trakcie analizy składniowej wejścia i nie ma potrzeby jawnego budowania drzewa wywodu (dzięki czemu zyskujemy na wydajności analizatora).

## Translacja sterowana składnią w metodzie zstępującej

- w ramach wykładu zademonstrowane zostanie konstruowanie analizatorów działających metodą rekurencyjnych zejść bez nawrotów najpierw w języku C, a następnie przy użyciu generatora LLgen
- gramatyki muszą być LL(1), jeśli nie są – należy:
  - wyeliminować lewostronną rekurencję
  - przeprowadzić lewostronną faktoryzację
  - odpowiednio dostosować akcje semantyczne



W ramach wykładu zostanie zademonstrowane konstruowanie analizatorów działających metodą rekurencyjnych zejść bez nawrotów najpierw w języku C, a następnie przy użyciu generatora LLgen.

Tego typu analizatory są efektywne, ale nakładają pewne istotne ograniczenia na gramatykę. Przed przystąpieniem do implementacji analizatora należy usunąć ewentualną lewostronną rekurencję i niejednoznaczności oraz – jeśli jest to konieczne – przeprowadzić lewostronną faktoryzację (zagadnienia te zostały szerzej omówione w wykładzie poświęconym analizie składniowej metodą zstępującą).

Podczas modyfikacji definicji sterowanych składnią trzeba również odpowiednio zmodyfikować akcje semantyczne.

W ramach wykładu poświęconego podstawom generatora LLgen omówiono jego rozszerzenia, które pozwalają obejść większość wymienionych ograniczeń.

## Interfejs do analizatora leksykalnego

- do analizy leksykalnej zostaną wykorzystane analizatory wygenerowane przez LEXa
- dla uproszczenia komunikacji ze skanerem warto odpowiednio rozszerzyć jego interfejs:

```
int LLcsymb;

int LLlookAhead(void)  { return LLcsymb; }

void LLread(void)   { LLcsymb = yylex(); }

void InitLexScanner(void)    { LLread(); }
```



W implementacji translatorów w języku C zostaną wykorzystane skanery wygenerowane za pomocą LEXa.

Dla uproszczenia komunikacji z analizatorem leksykalnym warto odpowiednio rozszerzyć jego interfejs o:

•pomocniczą zmienną *LLcsymb*, którą będzie przechowywać bieżący token

•funkcję *LLlookAhead* pozwalającą podejrzeć jeden token z wejścia

•funkcję *LLread* przesuwającą głowicę skanera na następny token

•funkcję *InitLexScanner* inicjalizującą skaner poprzez wczytanie pierwszego symbolu z wejścia.

## Atrybuty – zasady implementacji

- atrybuty symboli będą implementowane jako parametry funkcji:
  - atrybuty syntetyzowane jako parametry wyjściowe (na poziomie języka C – wskaźniki)
  - atrybuty dziedziczone jako parametry wejściowe (na poziomie języka C – zmienne przekazywane przez wartość)
- każdy symbol może mieć wiele atrybutów (ewentualne ograniczenia wynikają tylko z używanego kompilatora języka C)



Atrybuty symboli gramatyki będą implementowane jako parametry funkcji.

Atrybuty syntetyzowane jako parametry wyjściowe (na poziomie języka C – wskaźniki), atrybuty dziedziczone jako parametry wejściowe (na poziomie języka C – zmienne przekazywane przez wartość).

Z przyjętej metody implementacji wynika, że:

•każdy symbol może mieć wiele atrybutów (kompilator zgodny ze standardem C99 musi pozwalać na użycie co najmniej 127 parametrów funkcji)

•typy atrybutów syntetyzowanych i dziedziczonych podlegają tym samym zasadom co parametry funkcji w języku C.

Przedstawioną metodę można zastosować również w każdym innym języku programowania, który pozwala na korzystanie z podprogramów rekurencyjnych i przetwarzanie tekstu (np. Pascal i Ada, ale nie Basic i Fortan). Odpowiednie dostosowanie notacji i nazw typów nie powinno stanowić istotnego problemu.

Korzystanie z atrybutów syntetyzowanych i dziedziczonych zostanie zademonstrowane na przykładach.

## Atrybuty syntetyzowane – długość ciągu binarnego

- w implementacji analizatora obliczającego długość ciągu cyfr binarnych wykorzystany zostanie atrybut syntetyzowany
- do implementacji analizatora użyjemy następującej definicji sterowanej składnią:

```
S → L          { writeln(L.length) }
L → 0 L1       { L.length := L1.length + 1 }
L → 1 L1       { L.length := L1.length + 1 }
L → ε          { L.length := 0 }
```

- i analizatora leksykalnego o specyfikacji:

```
%%
[01]      { return yytext[0]; }
```



Wykorzystanie atrybutu syntetyzowanego zademonstrowane zostanie na przykładzie analizatora obliczającego i drukującego długość ciągu cyfr binarnych.

Jednostkowa produkcja S –> L posłuży do wypisania rezultatu.

Długość ciągu zostanie obliczona w atrybucie syntetyzowanym *length* nieterminala L, w którym po każdym natrafieniu na cyfrę binarną (0 lub 1) będziemy rekurencyjnie powtarzać rozpoznawanie aż do osiągnięcia przypadku bazowego – ciągu pustego. Ciąg pusty ma długość 0, a przy każdym powrocie z rekurencji wartość atrybutu *length* będzie zwiększana o 1.

Zadaniem analizatora leksykalnego będzie rozpoznawanie i zwracanie cyfr binarnych.

## Atrybuty syntetyzowane – nieterminal L

- implementacja nieterminala L:

```
void L(int *length)
{
  switch(LLlookAhead())
  {
     case '0' :
     case '1' : LLread();
                L(length);
                (*length)++;
                return;
     case  0  : *length = 0;
                return;
  }
}
```



Funkcja implementująca nieterminal L ma jeden parametr – wskaźnik na zmienną *length* (atrybut syntetyzowany).

Po każdym wywołaniu funkcja sprawdza symbol widoczny na wejściu. Po natrafieniu na cyfrę binarną głowica jest przesuwana na kolejny symbol (wywołaniem *LLread()*) i funkcja rekurencyjnie wywołuje samą siebie. Po powrocie z wywołania zostanie zwiększona wartość atrybutu *length*.

Po natrafieniu na koniec pliku (stała 0) atrybut *length* zostanie zainicjalizowany wartością 0 (długość ciągu pustego) i rozpocznie się powrót z rekurencji. Niestety, w etykiecie instrukcji *switch* nie można skorzystać ze stałej *EOF*, ponieważ w języku C ma ona wartość -1, a LEX sygnalizuje koniec pliku zwracając 0.

Wykorzystanie cech języka C umożliwiło łatwe wykonanie optymalizacji implementacji funkcji L. Ponieważ akcje semantyczne wykonywane po rozpoznaniu 0 i 1 były identyczne, więc można było wykorzystać ten sam kod w obu przypadkach (w Pascalu czy Adzie nie byłoby już to tak proste).

## Atrybuty syntetyzowane – nieterminal S

- implementacja nieterminala S i funkcji *main*:

```
void S(void)
{
  int len;
  L(&len);
  printf("\n%d\n",len);
}

int main()
{
  InitLexScanner();
  S();
  return 0;
}
```



W funkcji implementującej nieterminal S zadeklarowano lokalną zmienną *len*, która posłuży do akumulowania długości łańcucha. Wskaźnik do tej zmiennej jest przekazywany jako argument w wywołaniu funkcji L. Po powrocie z funkcji L drukujemy obliczoną przez nią długość łańcucha.

W funkcji *main* należy zainicjalizować analizator leksykalny, a następnie wywołać funkcję S, która jest implementacją aksjomatu gramatyki.

## Atrybuty dziedziczone – parzystość ciągu

- w implementacji analizatora sprawdzającego parzystość liczby binarnej wykorzystany zostanie atrybut syntetyzowany
- schemat translacji ma postać:

```
L → 0 { R.parity := 0 } R
L → 1 { R.parity := 1 } R
R → L
R → ε { if R.parity = 0
          then writeln("parzysta")
          else writeln("nieparzysta") }
```

- analizator leksykalny jest taki sam, jak w poprzednim przykładzie



Wykorzystanie atrybutu dziedziczonego zademonstrowane zostanie na przykładzie analizatora sprawdzającego parzystość liczby binarnej.

Na wejściu znajduje się niepusty ciąg cyfr binarnych tworzących liczbę zapisaną począwszy od najbardziej znaczącej cyfry.

Jeżeli liczba jest parzysta (najmłodszą cyfrą jest 0) ma zostać wydrukowany komunikat „parzysta”, w przeciwnym wypadku (najmłodszą cyfrą jest 1) – komunikat „nieparzysta”.

Wywodzenie wejścia rozpoczyna się od nieterminala L. Informacja o wczytanej cyfrze jest przekazywana w atrybucie dziedziczonym *parity* nieterminalowi R. Następnie rozwijany jest nieterminal R. Jeżeli na wejściu zostanie napotkany koniec pliku, na podstawie wartości atrybutu dziedziczonego drukowany jest odpowiedni komunikat. W przeciwnym wypadku – kontynuujemy wywodzenie nieterminala L.

Analizator leksykalny ma taką samą postać, jak w poprzednim przykładzie.

## Atrybuty dziedziczone – nieterminal R

- implementacja nieterminala R:

```
void R(int parity)
{
  switch(LLlookAhead())
  {
     case  0  : if(parity == 0)printf("parzysta");
                else printf("nieparzysta");
                return;
     default  : L();
                return;
  }
}
```



Funkcja implementująca nieterminal R ma jeden parametr – atrybut dziedziczony *parity* przekazywany przez wartość.

Jeżeli na wejściu widoczny jest koniec pliku, to na podstawie wartości tego atrybutu można określić jaka była ostatnio przeczytana cyfra i wydrukować odpowiedni komunikat. Jeżeli na wejściu widoczny jest jakikolwiek znak wywołujemy funkcję L.

## Atrybuty dziedziczone – nieterminal L

- implementacja nieterminala L:

```
void L(void)
{
  switch(LLlookAhead())
  {
     case '0' : LLread();
                R(0);
                return;
     case '1' : LLread();
                R(1);
                return;
  }
}
```



W implementacji nieterminala L należy wczytać cyfrę binarną z wejścia i przekazać odpowiednią informację w argumencie wywołania funkcji R.

Funkcja L jest implementacją aksjomatu gramatyki, więc to ona musi zostać wywołana w funkcji *main*.

## Atrybuty tokenów – suma liczb

- przekazywanie atrybutów tokenów zostanie pokazane na przykładzie prostego sumatora
- na wejściu znajduje się ciąg nieujemnych liczb całkowitych ujęty w nawiasy okrągłe,
  ciąg zawiera co najmniej jedną liczbę, jeśli jest ich więcej – są rozdzielone przecinkami
- należy napisać analizator, który obliczy i wypisze sumę liczb
- dla przykładowego wejścia *(5, 6, 7)* powinniśmy otrzymać odpowiedź: 18



Posługiwanie się atrybutami jednostek leksykalnych zademonstrujmy na przykładzie prostego sumatora.

Na wejściu znajduje się ciąg nieujemnych liczb całkowitych ujęty w nawiasy okrągłe. Ciąg zawiera co najmniej jedną liczbę, jeśli jest ich więcej – są rozdzielone przecinkami. Należy napisać analizator, który obliczy i wypisze sumę liczb.

## Atrybuty tokenów – schemat translacji

- wartość liczby jest przekazywana z analizatora leksykalnego za pomocą atrybutu *val*
- nieterminal R oblicza sumę liczb w atrybucie dziedziczonym *sum*
- schemat translacji ma postać:

```
S → ( num { R.sum := num.val } R

R → ) { writeln(R.sum) }
R → , num { R1.sum := R.sum + num.val } R1
```



W translacji wykorzystany zostanie atrybut (oczywiście syntetyzowany) *val* symbolu terminalnego *num*, który posłuży do przekazania wartości liczby do analizatora składniowego.

Całe wejście (S) składać się będzie z lewego nawiasu, liczby i reszty (R). Po odczytaniu liczby jej wartość (atrybut *val*) kopiujemy do atrybutu dziedziczonego nieterminala R.

Reszta (R) może być:

•prawym nawiasem kończącym wejście – wtedy drukujemy rezultat (atrybut dziedziczony *sum* nieterminala R)

•przecinkiem, kolejną liczbą i resztą (R1) – do atrybutu *sum* dalszego ciągu reszty przypisujemy sumę dotychczas przetworzonych elementów (atrybut *sum* nieterminala R) i wartość bieżącej liczby (*num.val*).

## Atrybuty tokenów – skaner

- analizator leksykalny:

```
%{ extern int LLlval;
   #define num 257
%}
%%
\(          { return '('; }
\)          { return ')'; }
\,          { return ','; }
" "         { ; }
[0-9]+      { LLlval = atoi(yytext);
              return num;
            }
```



Analizator leksykalny zaimplementujemy w LEXie. Musi on rozpoznawać i zwracać lewe i prawe nawiasy oraz przecinek. Ewentualne spacje na wejściu pomijamy.

Do przekazywania wartości atrybutu symbolu leksykalnego musimy wykorzystać globalną zmienną – w tym przypadku wystarczy użyć zmiennej typu całkowitego. Jeżeli będziemy przekazywać atrybuty różnych typów można skorzystać z unii języka C.

Zmienną możemy zdefiniować w parserze albo w skanerze. W przykładzie zmienna *LLlval* została zdefiniowana w parserze, a więc w skanerze musimy zadeklarować ją jako zewnętrzną. Po natrafieniu na liczbę, jej wartość wpisujemy do zmiennej *LLlval* i zwracamy informację o rozpoznaniu tokenu *num*.

Tokenowi *num* przypisaliśmy nazwę, więc musimy pamiętać o przypisaniu jej odpowiedniej (większej niż 256) stałej. Gdyby w gramatyce było więcej nazwanych tokenów najlepiej utworzyć zewnętrzny plik nagłówkowy i włączać go i do analizatora leksykalnego i składniowego.

## Atrybuty tokenów – nieterminal S

- w parserze definiujemy globalną zmienną:

  ```
  int LLlval;
  ```

  oraz token *num*:

  ```
  #define num 257
  ```

- implementacja nieterminala S ma postać:

```
void S(void)
{
  LLread(); /* ( */
  LLread(); /* num */
  R(LLlval);
}
```



Zmienną *LLlval* służącą do przekazywania atrybutu syntetyzowanego symbolu *num* definiujemy w analizatorze składniowym.

Musimy pamiętać również o przypisaniu stałej dla tokenu *num* – oczywiście o tej samej wartości co w analizatorze leksykalnym.

Funkcja implementująca nieterminal S czyta z wejścia lewy nawias, liczbę i wywołuje funkcję R przekazując jako argument wartość liczby (atrybut dziedziczony *sum*).

## Atrybuty tokenów – nieterminal R

- implementacja nieterminala R:

```
void R(int sum)
{ switch(LLlookAhead())
  {  case ')' : LLread();
                printf("\n%d\n",sum);
                return;
     case ',' : LLread(); /* , */
                LLread(); /* num */
                R(sum + LLlval);
                return;
  }
}
```



Funkcja implementująca nieterminal R podgląda jeden symbol z wejścia. Jeżeli jest to nawias, wczytuje go i drukuje wartość sumy.

Jeżeli jest to przecinek, wczytuje go, wczytuje liczbę i wywołuje procedurę R przekazując jako argument sumę wcześniej wczytanych (zmienna *sum*) i wartość bieżącej liczby (*LLlval*).

## Obsługa błędów - zadanie

- problem obsługi błędów zostanie omówiony na przykładzie akceptora dla kontekstowego języka $a^n b^n c^n$ dla $n > 0$
- cała obsługa błędów będzie wykonywana po stronie analizatora składniowego
- analizator leksykalny rozpoznaje i zwraca pojedyncze znaki:

```
%%
.|\n      { return yytext[0]; }
```



We wszystkich przedstawionych wcześniej analizatorach zakładaliśmy, że wejście jest poprawne. Zobaczmy teraz na przykładzie akceptora dla kontekstowego języka a^n b^n c^n jak można zorganizować obsługę błędów w analizatorze składniowym.

Analizator leksykalny rozpoznaje dowolne znaki na wejściu i przekazuje je do parsera, na którego spada całe zadanie obsługi błędów. Takie rozwiązanie jest o wiele efektywniejsze, jeśli chodzi o wykrywanie możliwych przyczyn błędu – tylko parser wie, w którym miejscu, który znak jest dozwolony i dzięki temu jest w stanie wydrukować bardziej deskryptywny komunikat o błędzie i podjąć rozsądną próbę kontynuacji.

## Obsługa błędów – gramatyka

- definicja sterowana składnią – nieterminal S:

```
S → A B C      { if A.len = B.len and
                      B.len = C.len
                   then writeln('OK')
                   else writeln('Error')
```

- definicja sterowana składnią – nieterminal A:

```
A  → a RA      { A.len := RA.length + 1 }

RA → a RA1     { RA.length:=RA1.length + 1; }
RA → ε         { RA.length:=0 }
```



Język wejściowy jest językiem kontekstowym, więc nie da się go opisać gramatyką LL(1). Musimy zdefiniować gramatykę bezkontekstową opisującą szerszy język (zgodny z wyrażeniem regularnym a+b+c+), a następnie za pomocą akcji semantycznych sprawdzić, czy wejście jest poprawne.

Całe wejście (S) składa się z ciągu liter a (nieterminal A), ciągu liter b (nieterminal B) i ciągu liter c (nieterminal C). Nieterminale A, B i C mają atrybuty syntetyzowane *len*, w których będzie akumulowana długość ciągów poszczególnych liter. Po przetworzeniu całego wejścia porównywane są długości poszczególnych ciągów, jeśli są równe drukowany jest komunikat „OK”, jeśli są różne – komunikat „Error”.

Ciąg liter *a* (opisany nieterminalem A) składa się z pojedynczej litery *a* i reszty (RA). Rezultatem (atrybutem syntetyzowanym *len* symbolu A) będzie więc długość reszty (atrybutu *length* symbolu RA) powiększona o 1.

Reszta (RA) to albo litera *a* i dalszy ciąg reszty (RA1), albo napis pusty. W pierwszym przypadku rezultatem (atrybut *length* nieterminala RA) będzie długość reszty (atrybut *length* symbolu RA1) powiększona o 1, w drugim przypadku – bazowym – 0.

Nieterminale B (RB) i C (RC) mają (z dokładnością do rozpoznawanych znaków) identyczną definicję.

## Obsługa błędów – rozbudowa skanera

- do realizacji obsługi błędów przydatna będzie możliwość wycofania do strumienia wejściowego zadanego tokenu
- niezbędne jest rozbudowanie interfejsu o:
  - dodatkową zmienną *LLusymb* przechowującą wycofany token:
    ```
    int LLusymb = -1;
    ```
  - funkcję *LLpushBack* wycofującą token:
    ```
    void LLpushBack(int token)
    { LLusymb = token; }
    ```



Obsługa błędów wymaga rozbudowania interfejsu między parserem a skanerem o możliwość wycofania do strumienia wejściowego zadanego tokenu.

Potrzebna będzie dodatkowa zmienna (*LLusymb*) do przechowywania wycofanego tokenu oraz funkcja (LLpushback) wycofująca token podany jako argument wywołania.

## Obsługa błędów – modyfikacje skanera

- niektóre dotychczas używane funkcje również wymagają odpowiednich modyfikacji:

```
int LLlookAhead(void)
{
  if(LLusymb == -1)return LLcsymb;
  else return LLusymb;
}
void LLread(void)
{
  if(LLusymb != -1)LLusymb = -1;
  else LLcsymb = yylex();
}
```



Używane wcześniej funkcje *LLlookAhead* i *LLread* należy odpowiednio zmodyfikować. Funkcja *LLInitLexScanner* nie wymaga modyfikacji.

## Obsługa błędów – aksjomat S

- implementacja nieterminala S:

```
void S(void)
{
  int l_a, l_b, l_c;
  A(&l_a);
  B(&l_b);
  C(&l_c);
  if((l_a == l_b) &&
     (l_b == l_c))puts("OK");
  else printf("Error");
}
```



W implementacji nieterminala S (aksjomatu gramatyki):

•deklarujemy zmienne *l_a*, *l_b*, *l_c*, w których przechowywana będzie wartość atrybutów syntetyzowanych *len* nieterminali A, B i C

•kolejno wywołujemy funkcje implementujące nieterminale A, B i C podając jako parametry wskaźniki do odpowiednich zmiennych

•po przetworzeniu wejścia porównujemy długości ciągów i drukujemy odpowiedni komunikat.

## Obsługa błędów – nieterminal A

- implementacja nieterminala A:

```
void A(int *len)
{
  int length;
  switch(LLlookAhead())
  {
    case 'a' : LLread(); RA(&length); (*len)=++length;
               return;
    case 'b' :
    case 'c' : puts("Oczekiwane a");
               return;
    case  0  : puts("Nieoczekiwany koniec pliku");
               exit(1);
    default  : printf("Nieoczekiwany znak {%c}",LLlookAhead());
               exit(1);
  }
}
```



W trakcie implementowania funkcji dla każdego nieterminala osobno należy rozważyć akcje dla poprawnego wejścia i dla błędów.

Jakość detekcji i sygnalizacji błędów będzie zależeć od staranności analizy przeprowadzonej na tym etapie. Najprostszym rozwiązaniem jest oczywiście natychmiastowe przerwanie działania z komunikatem o błędzie. Ilość i jakość informacji o błędzie jaka zostaje przekazana użytkownikowi jest wtedy minimalna.

Dodanie kodu diagnostyki błędów zdecydowanie powiększa rozmiar analizatora, co widać w przedstawionym przykładzie (kod zaznaczony na czerwono służy do obsługi błędów). W analizatorze przedstawiono jedną z możliwych interpretacji błędów – nie w każdym przypadku da się jednoznacznie określić na czym polegał i w którym miejscu wystąpił błąd.

Funkcja implementująca nieterminal A podgląda jeden symbol z wejścia, którym może być:

•litera a – wejście jest poprawne, przesuwamy głowicę, wywołujemy funkcję RA w celu obliczenia długości reszty ciągu, po powrocie zwiększamy długość o 1 (znak, który odczytaliśmy przed wywołaniem funkcji)

•litery b i c – wejście jest błędne, zabrakło litery a, drukujemy komunikat diagnostyczny i powracamy z funkcji

•koniec pliku – wejście jest puste, dalsza diagnostyka nie ma sensu, drukujemy komunikat i przerywamy program

•dowolny inny znak – w pliku są znaki spoza zbioru rozpoznawanych znaków, również nie ma sensu kontynuować analizy, drukujemy komunikat i przerywamy program

## Obsługa błędów – nieterminal RA

- implementacja nieterminala RA:

```
void RA(int *length)
{
  switch(LLlookAhead())
  {
    case 'a' : LLread(); RA(length); (*length)++;
               return;
    case 'b' : (*length)=0;
               return;
    case 'c' : puts("Oczekiwane b"); LLpushBack('b');
               return;
    case  0  : puts("Nieoczekiwany koniec pliku");
               exit(1);
    default  : printf("Nieoczekiwany znak {%c}",LLlookAhead());
               exit(1);
  }
}
```



Funkcja implementująca nieterminal RA podgląda jeden symbol z wejścia, którym może być:

•litera a – wejście jest poprawne, przesuwamy głowicę, wywołujemy rekurencyjnie funkcję RA w celu obliczenia długości reszty ciągu, po powrocie zwiększamy długość o 1 (znak, który odczytaliśmy przed wywołaniem funkcji)

•litera b – poprawne wejście (Follow(A)), zakończyliśmy analizę łańcucha liter a i rozpoczynamy powrót do aksjomatu S

•litera c – wejście jest błędne, zabrakło litery b, drukujemy komunikat diagnostyczny i aby móc kontynuować analizę po powrocie z A naprawiamy wejście i powracamy z funkcji

•koniec pliku – wejście jest puste, dalsza diagnostyka nie ma sensu, drukujemy komunikat i przerywamy program

•dowolny inny znak – w pliku są znaki spoza zbioru rozpoznawanych znaków, nie ma sensu kontynuować analizy, drukujemy komunikat i przerywamy program

## Obsługa błędów – nieterminal B

- implementacja nieterminala B:

```
void B(int *len)
{
  int length;
  switch(LLlookAhead())
  {
    case 'b' : LLread();
               RB(&length);
               (*len)=++length;
               return;
    case 'c' : puts("Oczekiwane b");
               return;
    /*   case  0  : obsłużony w RA */
    default  : printf("Nieoczekiwany znak {%c}",LLlookAhead());
               exit(1);
  }
}
```



Funkcja implementująca nieterminal B podgląda jeden symbol z wejścia, którym może być:

•litera b – wejście jest poprawne, przesuwamy głowicę, wywołujemy funkcję RB w celu obliczenia długości pozostałej części łańcucha a po powrocie zwiększamy długość o 1 (znak, który odczytaliśmy przed wywołaniem funkcji)

•litera c – wejście jest błędne, zabrakło litery b, drukujemy komunikat diagnostyczny i powracamy z funkcji

•koniec pliku – błąd, który został już obsłużony w funkcji RA

•dowolny inny znak – w pliku są znaki spoza zbioru rozpoznawanych znaków, nie ma sensu kontynuować analizy, drukujemy komunikat i przerywamy program

## Obsługa błędów – nieterminal RB

- implementacja nieterminala RB:

```
void RB(int *length)
{
  switch(LLlookAhead())
  {
    case 'b' : LLread();
               RB(length);
               (*length)++;
               return;
    case 'c' : (*length)=0;
               return;
    case  0  : puts("Nieoczekiwany koniec pliku");
               exit(1);
    default  : printf("Nieoczekiwany znak {%c}",LLlookAhead());
               exit(1);
  }
}
```



Procedura RB podgląda jeden symbol z wejścia, którym może być:

•litera b – wejście jest poprawne, przesuwamy głowicę, wywołujemy rekurencyjnie funkcję RB w celu obliczenia długości dalszego ciągu łańcucha, po powrocie zwiększamy długość o 1 (znak, który odczytaliśmy przed wywołaniem funkcji)

•litera c – poprawne wejście (Follow(B)), zakończyliśmy analizę łańcucha liter b i rozpoczynamy powrót do aksjomatu S

•koniec pliku – wejście jest puste, dalsza diagnostyka nie ma sensu, drukujemy komunikat i przerywamy program

•dowolny inny znak – w pliku są znaki spoza zbioru rozpoznawanych znaków, kontynuowanie analizy jest bezcelowe, drukujemy komunikat i przerywamy program

## Obsługa błędów – nieterminal C

- implementacja nieterminala C:

```
void C(int *len)
{
  int length;
  switch(LLlookAhead())
  {
    case 'c' : LLread();
               RC(&length);
               (*len)=++length;
               return;
    /*   case  0  : obsłużony w RB */
    default  : printf("Nieoczekiwany znak {%c}",LLlookAhead());
               exit(1);
  }
}
```



Procedura C podgląda jeden symbol z wejścia, którym może być:

•litera c – wejście jest poprawne, przesuwamy głowicę, wywołujemy funkcję RC w celu obliczenia długości dalszego ciągu łańcucha, po powrocie zwiększamy długość o 1 (znak, który odczytaliśmy przed wywołaniem funkcji)

•koniec pliku – błąd, który został już obsłużony w RB

•dowolny inny znak jest błędem – drukujemy komunikat i przerywamy program

## Obsługa błędów – nieterminal RC

- implementacja nieterminala RC:

```
void RC(int *length)
{
  switch(LLlookAhead())
  {
    case 'c' : LLread();
               RC(length);
               (*length)++;
               return;
    case 0   : (*length)=0;
               return;
    default  : printf("Nieoczekiwany znak {%c}",LLlookAhead());
               exit(1);
  }
}
```



Procedura RC podgląda jeden symbol z wejścia, którym może być:

•litera c – wejście jest poprawne, przesuwamy głowicę, wywołujemy rekurencyjnie funkcję RC w celu obliczenia długości dalszego ciągu łańcucha, po powrocie zwiększamy długość o 1 (znak, który odczytaliśmy przed wywołaniem funkcji)

•koniec pliku – poprawne wejście (Follow(C)), zakończyliśmy analizę łańcucha liter c i rozpoczynamy powrót do aksjomatu S

•dowolny inny znak jest błędem – przerywamy analizę, drukujemy komunikat i przerywamy program

## LLgen – implementacja atrybutów

- w generatorze LLgen atrybuty symboli są implementowane w identyczny sposób jak w języku C (jako parametry funkcji ):
  - atrybuty syntetyzowane jako parametry wyjściowe (na poziomie języka C – wskaźniki)
  - atrybuty dziedziczone jako parametry wejściowe (na poziomie języka – C zmienne przekazywane przez wartość)



W generatorze *LLgen* atrybuty symboli są implementowane w dokładnie taki sam sposób jak w języku C.

Atrybuty są implementowane jako parametry funkcji:

•atrybuty syntetyzowane jako parametry wyjściowe (na poziomie języka C – wskaźniki)

•atrybuty dziedziczone jako parametry wejściowe (na poziomie języka C – zmienne przekazywane przez wartość)

## LLgen – implementacja atrybutów

- generator LLgen nie nakłada jawnych ograniczeń na liczbę i typ atrybutów symboli
- np. nieterminal *spec* w produkcji:

```
spec(int p1; int p2;
     int *p3; double *p4;)
     { int l1; double l2; } : ...
```

ma:
2 atrybuty dziedziczone (p1 i p2)
2 atrybuty syntetyzowane (p3 i p4)
2 zmienne lokalne (l1 i l2)



Generator LLgen nie nakłada własnych, dodatkowych ograniczeń na liczbę i typ atrybutów ponad te, które wynikają z użycia języka C.

Atrybuty i zmienne lokalne deklarowane są w produkcji, w której dany nieterminal znajduje się po lewej stronie (jest tylko jedna taka produkcja dla każdego nieterminala, ponieważ w LLgenie alternatywne prawe strony muszą być zapisywane po znaku ‘|’).

Atrybuty definiujemy zaraz po nazwie symbolu, w nawiasach okrągłych, rozdzielając je średnikami, średnik po ostatnim atrybucie jest opcjonalny. Zmienne lokalne deklarujemy po nazwie symbolu i atrybutach – w nawiasach klamrowych.

Użycie atrybutów w LLgenie zostanie pokazane na tych samych przykładach, które były wykorzystane wcześniej, przy implementacji analizatorów w języku C. Pozwoli to na porównanie o ile wykorzystanie generatora pozwala uprościć implementację translatora.

Nie zostaną omówione zasady obsługi błędów, analiza jaką trzeba przeprowadzić jest dokładnie taka sama, jak w przypadku implementacji w języku C, a implementacja funkcji *LLmessage* służącej do obsługi błędów została omówiona w wykładzie poświęconym podstawom generatora LLgen.

## LLgen – atrybuty syntetyzowane

- specyfikacja analizatora obliczającego długość ciągu binarnego:

```
%start parse, S ;

S { int len; }
  : L(&len)        { printf("\n[%d]\n",len); }
  ;
L(int *length)
  : '0' L(length) { (*length)++; }
  | '1' L(length) { (*length)++; }
  |               { *length = 0; }
  ;
```



Wykorzystanie atrybutu dziedziczonego w LLgenie można zademonstrować na przykładzie obliczania długości ciągu binarnego (problem i schemat translacji zostały przedstawione na slajdzie 11.).

Nieterminal S jest aksjomatem gramatyki, ma zmienną lokalną *len*, która posłuży do przechowywania długości ciągu w trakcie obliczeń. Służy jedynie do rozwinięcia nieterminala L i wydrukowania obliczonego w nim rezultatu.

Nieterminal L ma atrybut dziedziczony *length*. Po napotkaniu cyfry binarnej rekurencyjnie wywołuje sam siebie, a po powrocie zwiększa długość o 1. Przypadkiem bazowym jest ciąg pusty – długość ciągu wynosi wtedy 0.

## LLgen – atrybuty syntetyzowane

- alternatywne specyfikacje nieterminala L:

**tradycyjna notacja:**

```
L(int *length)
  : '0' L(length) { (*length)++; }
  | '1' L(length) { (*length)++; }
  |               { *length = 0; }
  ;
```

**wykorzystanie rozszerzeń LLgena:**

```
L(int *length)
  : ['0' | '1'] L(length) { (*length)++; }
  |                       { *length = 0; }
  ;
```



Warto zauważyć jakie możliwości uproszczenia zapisu dają rozszerzenia składni wprowadzone do generatora LLgen.

W przykładzie zestawiono tradycyjny zapis produkcji dla nieterminala L z wersją, w której wykorzystano rozszerzenia, dzięki którym zapis specyfikacji jest krótszy, a przy okazji zmniejszył się kod programu wynikowego.

## LLgen – atrybuty dziedziczone

- specyfikacja analizatora badającego parzystość liczby binarnej:

```
%start parse, S ;
S : '0' R(0)
  | '1' R(1)
  ;
R(int parity)
  : S
  | { if(parity == 1)puts("even");
      else puts("odd");
    }
  ;
```



Wykorzystanie atrybutu dziedziczonego w LLgenie można zademonstrować na przykładzie badania parzystości liczby binarnej (problem i schemat translacji zostały przedstawione na slajdzie 14.).

Nieterminal S (aksjomat gramatyki) wczytuje z wejścia cyfrę binarną i przekazuje informację o niej jako atrybut dziedziczony nieterminala R.

Nieterminal R, jeżeli natrafi na ciąg pusty (koniec pliku) drukuje na postawie atrybutu *parity* odpowiedni komunikat, a w przeciwnym przypadku powraca do rozwijania nieterminala S.

## LLgen – atrybuty tokenów

- obliczanie sumy liczb – specyfikacja skanera:

```
%{
  #include <stdio.h>
  extern int token_val;
  #include "Lpars.h"
%}
%%
\(          { return '('; }
\)          { return ')'; }
\,          { return ','; }
" "         { return ' '; }
[0-9]+      { token_val = atoi(yytext);
              return num;
            }
```



Posługiwanie się atrybutami tokenów pokażemy na przykładzie obliczania sumy liczb (problem i schemat translacji zostały przedstawione na slajdzie 17.).

LLgen nie oferuje żadnego systemowego rozwiązania problemu przekazywania atrybutów tokenów. Problem ten musimy więc rozwiązać samodzielnie za pomocą zmiennych globalnych. Jest więc to takie samo podejście jakie zostało wykorzystane w implementacji translatorów za pomocą języka C.

W analizatorze leksykalnym deklarujemy zewnętrzną zmienną *token_val* służącą do przekazywania wartości rozpoznanej liczby (atrybutu terminala *num*).

LLgen ulży nam natomiast w kwestii przypisywania stałych nazwanym tokenom – wystarczy odpowiednie deklaracje umieścić w specyfikacji analizatora składniowego, a do skanera włączyć plik interfejsu Lpars.h.

## LLgen – atrybuty tokenów

- obliczanie sumy liczb – specyfikacja pasera:

```
{
  int token_val;
}

%token num;

%start parse, S ;

S : '(' num  R(token_val)
  ;
R(int sum) : ')' { printf("suma = %d", sum); }
           | ',' num R(sum + token_val)
           ;
```



Specyfikacja analizatora składniowego zawiera deklarację globalnej zmiennej *token_val*, deklarację tokenu *num* oraz aksjomatu i nazwy funkcji implementującej aksjomat.

Wejście (nieterminal S) składa się z lewego nawiasu, liczby i reszty (R). Po odczytaniu liczby jej wartość (zmienna *token_val*) jest przekazywana jako atrybut dziedziczony nieterminala R.

Reszta (R) może być:

•prawym nawiasem kończącym wejście – wtedy drukujemy rezultat (atrybut dziedziczony *sum* nieterminala R)

•przecinkiem, kolejną liczbą i resztą – reszta jako argument otrzymuje sumę dotychczas przetworzonych elementów i wartość bieżącej liczby (*token_val*).

## Dalsza lektura

- translacja sterowana składnią:
  - Aho A. V., Sethi R., Ullman J. D., Compilers: Principles, Techniques, and Tools, Addison-Wesley, 1986
  - Waite W. M., Goos G., Konstrukcja kompilatorów, WNT 1989
- generator LLgen:
  - Grune D., Jacobs C. J. H., A Programmer-friendly LL(1) Parser Generator, Software - Practice and Experience, vol. 18, no. 1, pp. 29-38, 1/1988
  - Jacobs C. J. H., LLgen, an extended LL(1) parser generator, http://tack.sourceforge.net/doc/LLgen.html