Języki formalne i kompilatory

(15 wyk., 15 ćw. audyt., 15 ćw. lab.)

dr inż. **Jolanta Cybulka**

Instytut Automatyki, Robotyki i Inżynierii Informatycznej, pok. 313A Piotrowo bud. A1

e-mail: Jolanta.Cybulka@put.poznan.pl

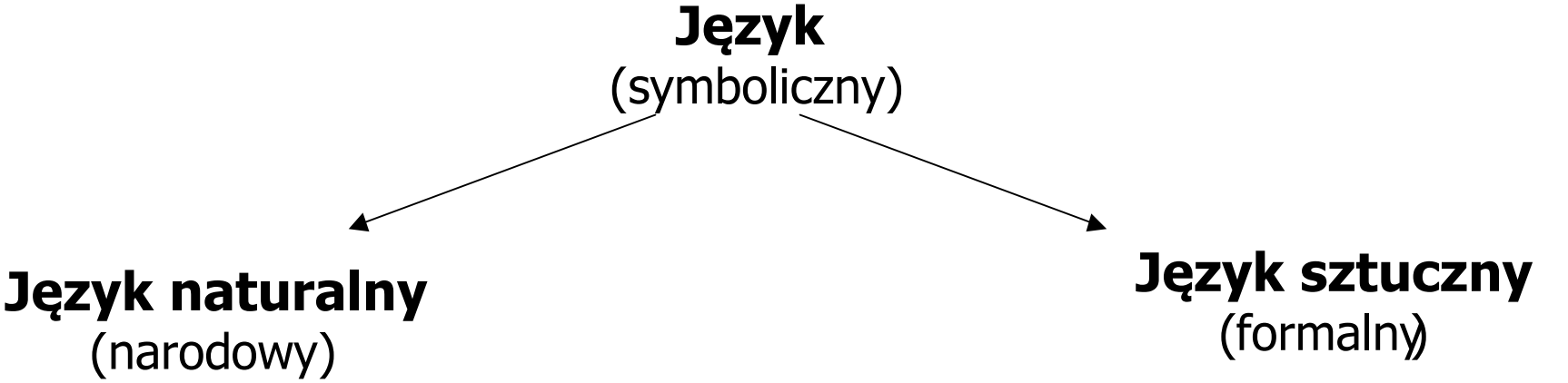
www: http://users.man.poznan.pl/jolac

Literatura

1. Aho A V., Sethi R., Ullman J.: Kompilatory. Reguły metody i narzędzia. WNT, Warszawa 2002.
2. Cybulka J., Jankowska B., Nawrocki J. R.: Automatyczne przetwarzanie tekstów. AWK, Lex i YACC, Wyd. NAKOM, Poznań, 2002.
3. Dembiński P., Małuszyński J.: Matematyczne metody definiowania języków programowania, WNT, Warszawa 1981.
4. Hopcroft J.E., Ullman J.D.: Wprowadzenie do teorii automatów, języków i obliczeń, PWN, Warszawa, 1994.
5. Révész G.E.: Introduction to Formal languages, McGraw-Hill, New York, 1983.

Spis treści

1. Języki formalne – podstawowe pojęcia



- rozważamy ***języki symboliczne***, skonstruowane z symboli należących do zbioru (niepustego, skończonego), zwanego ***alfabetem***, np.:

  D = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}

  B = {0,1}

  H = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F}

  Lat = {a, b, c, …, z, A, B, C,…, Z}

  Pol = {a, ą, b, c, ć, …, ż, dz, dź, dż, A, Ą, B, C, Ć,…, Z, Dz, Dź, Dż}

  S = {¬, ∧, ∨, ⇔, ⇒, p, f, (,)}

  i wiele innych (alfabet Morse'a, Braille'a, symbole nutowe, symbole chemiczne, znaki migowe,…);

- z symboli alfabetu, zgodnie z zadanymi ***regułami składania*** symboli powstają ***napisy języka*** – zbiór wszystkich napisów należących do języka (i tylko takich) nazywamy jego ***składnią*** (syntaktyką, syntaksą);

0 123 09 55000 nad alfabetem **D**

0001 11001 0 1 nad alfabetem **B**

0 123F 09A FFAA nad alfabetem **H**

Fundamenta Informaticae Formal Language nad alfabetem **Lat**

Ala mieć ma As Asa nad alfabetem **Pol**

p $\wedge$ f p (f $\vee$ p) $\wedge$ p $\Leftrightarrow$ p nad alfabetem **S**

- składnia może mieć strukturę warstwową, tzn. z ciągu symboli powstają napisy elementarne, których zbiór staje się nowym alfabetem, z którego powstają nowe napisy:

nad alfabetem **L = {Ala, mieć, ma, As, Asa, \ , \n, \t…}**
można zbudować napis **Ala ma Asa**

nad alfabetem **C =** {**if**, **else**, **(**, **)** , **{**, **}**,**=**, **;**, **++**, >, **num**, **id** , **\ , \n, \t…**}
można zbudować napis **if (xsr >12) {y++;} else {y=xsr;}**

- w wypadku języków formalnych, na składni można wykonywać operacje mnogościowe, jak: ***suma***, ***różnica***, ***przecięcie***, ***dopełnienie*** do uniwersum, ***złożenie***, ***potęgowanie***, ***domknięcie***;

  $A = \{a, b, c\}$ alfabet

  – potęgowanie: A^0, A^1, A^2, A^3, …

  $A^0 = \{\varepsilon\}$ słowo długości 0, słowo „puste”

  $A^1 = \{ a, b, c \}$ słowa długości 1

  $A^2 = \{ aa, ab, ac, ba, bb, bc, ca, cb, cc \}$ słowa długości 2

  $A^3 = \{ aaa, aab, aac, aba, abb, abc, aca, acb, acc, baa, bab, bac, bba, bbb, bbc, bca, bcb, bcc, caa, cab, cac, cba, cbb, cbc, cca, ccb, ccc\}$ słowa długości 3

  …

  – domknięcie zwrotne i przechodnie operacji potęgowania: A^*

  $$A^* = A^0 \cup A^1 \cup A^2 \cup A^3 \cup \ldots = \bigcup_{(i = 0,\infty)} A^i$$

  domknięcie przechodnie operacji potęgowania: A^+

  $$A^+ = A^1 \cup A^2 \cup A^3 \cup \ldots = \bigcup_{(i = 1,\infty)} A^i$$

- napisy (składnia) służą do notowania znaczeń, zadaniem funkcji semantyki języka formalnego jest przypisanie każdemu elementowi składniowemu (napisowi) znaczenia z pewnej dziedziny znaczeń:

znaczeniem napisu **(2+3)*4** nad alfabetem $\mathbf{A_n}$ = {**0**,**1**,…,**9**}$\cup$ $\mathbf{A_o}$ ={**+**, ***** }$\cup${ **(,)**}
jest **20** o ile

D = {**1**, **2**, **3**, …} jest zbiorem wartości (liczb) naturalnych,

F = {**plus**, **times**} jest zbiorem funkcji (dodawania i mnożenia),

$\mathbf{val_n : A_n \rightarrow D}$

$\mathbf{val_o : A_o \rightarrow F}$, $\mathbf{val_o}$(**+**)= **plus**, $\mathbf{val_o}$(*****)= **times**

value : $(\mathbf{A_n \cup A_o \cup}$ { **(,)**})***** $\rightarrow$ **D** (dla napisów poprawnych składniowo)

- język formalny L definiujemy matematycznie jako:

$$\mathbf{L} = \langle \Sigma, \mathbf{Syn}, \mathbf{DSem}, \mathbf{Sem} \rangle$$

Σ – ***alfabet***, skończony niepusty zbiór symboli (znaków),

Syn – ***syntaktyka*** (składnia, syntaksa), zbiór napisów języka, zbudowanych z symboli alfabetu

DSem – ***dziedzina znaczeń***, zbiór bytów semantycznych, przypisywanych napisom języka

Sem – ***relacja semantyczna*** (**Sem** $\subseteq$ **Syn** $\times$ **DSem**), definiująca związki między napisami języka a elementami dziedziny znaczeń (relacja ta w zastosowaniach praktycznych jest ***funkcją semantyki***);

- dodatkowo, język zastosowany w pewnej dziedzinie może wytwarzać odmiany ***pragmatyczne***:

$$\mathbf{L_1} = \langle \Sigma, \mathbf{Syn}, \mathbf{DSem_1}, \mathbf{Sem_1} \rangle \qquad \mathbf{L_2} = \langle \Sigma, \mathbf{Syn}, \mathbf{DSem_2}, \mathbf{Sem_2} \rangle$$

Na czym może polegać ***definiowanie i przetwarzanie języka formalnego***?

a) określenie języka (wszystkich jego elementów) w sposób ścisły, za pomocą odpowiednich metod i narzędzi

b) badanie przynależności napisu do języka (sprawdzenie poprawności napisu)

c) translacja napisu z jednego języka na drugi (przy zachowaniu znaczenia)

2. Języki regularne (klasa "3")

Wyrażenia regularne

- ***wyrażeniem regularnym*** (ang. *regular expression*) nad alfabetem Σ nazywamy napis zbudowany za pomocą następującej rekurencyjnej definicji:
  1) $\varnothing$ jest w. r. oznaczającym język pusty (pusty zbiór napisów)
  2) ε jest w. r. oznaczającym język $\{\varepsilon\}$
  3) jeśli $a \in \Sigma$, to napis a jest w. r. oznaczającym język $\{a\}$
  4) jeśli r_1 i r_2 są w. r., to napis $r_1 r_2$ jest w. r. oznaczającym język $L(r_1)L(r_2)$
  5) jeśli r_1 i r_2 są w. r., to napis $r_1|r_2$ jest w. r. oznaczającym język $L(r_1) \cup L(r_2)$
  6) jeśli r jest w. r., to napis r^* jest w. r. oznaczającym język $\cup_{(i=0,\infty)} L(r)^i$
  7) jeśli r jest w. r., to (r) jest w. r. oznaczającym $L(r)$.

- właściwości wyrażeń regularnych; niech p, q i r oznaczają dowolne wyrażenia regularne, wtedy:

$p \mid (q \mid r) = (p \mid r) \mid q$	$p(qr) = (pq)r$
$p \mid q = q \mid p$	$p^* = \varepsilon \mid pp^*$
$(p \mid q)r = pr \mid qr$	$r(p \mid q) = rp \mid rq$
$\varepsilon p = p\varepsilon = p$	$p^* = (\varepsilon \mid p)^*$
$\varnothing p = p\varnothing = \varnothing$	if $p = r \mid pq \wedge q \neq \varepsilon$ then $p = rq^*$

- **przykłady**

  Niech $\Sigma = \{0, 1, 2\}$

  1) 11 reprezentuje język $\{11\}$
  2) $(0 \mid 1)^*$ reprezentuje język wszystkich napisów zbudowanych z zer i jedynek
  3) $(0 \mid 1)^*11(0 \mid 1)^*$ reprezentuje język wszystkich napisów zbudowanych z zer i jedynek, w których co najmniej raz występują dwie kolejne jedynki
  4) $0^*1^*2^*$ reprezentuje język składający się z napisów zbudowanych z dowolnej liczby zer, po której następuje dowolna liczba jedynek, a następnie dowolna liczba dwójek;

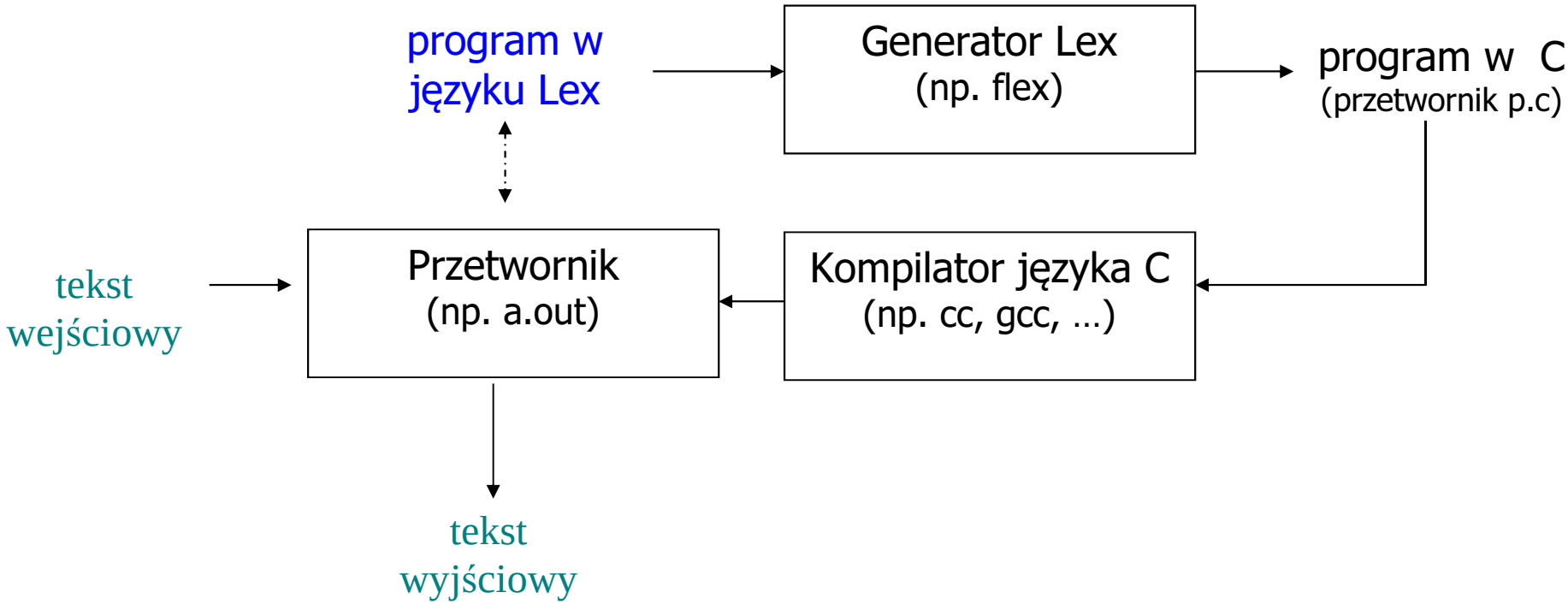
Niech Σ = {0, 1, 2, …9, a, b,…z, A, B…Z, _, –, +}

5) (a|b|…z|A|B…Z)(a|b|…z|A|B…Z|0|1|…9)* reprezentuje język niepustych ciągów liter i cyfr, rozpoczynających się od litery

6) (-|+|ε)(0|1|…9)(0|1|…9)* reprezentuje język stałych całkowitych dziesiętnych

2.2 Język Lex

- ***programy*** w języku Lex ***opisują działanie przetworników tekstu***
- konkretny system Lex (np. lex, flex) jest generatorem przetworników zdefiniowanych w języku Lex:



- ***struktura programu*** w języku Lex:

definicje pomocnicze		opcjonalne (dyrektywy, nazwy wyrażeń, deklaracje, …)
%%		
wzorzec 1	*akcja*1	obligatoryjne (reguły przetwarzania)
wzorzec 2	*akcja*2	
…		
wzorzec n	*akcja* n	
%%		
podprogramy pomocnicze		opcjonalne

2.2.1 Wzorce w programach

- mają postać (poszerzonych o nowe operatory) ***definicji regularnych*** (w cudzysłowie lub bez)
- definicje regularne są zbudowane nad pewnym ***alfabetem*** oraz zbiorem ***nazw*** (nazw wyrażeń regularnych)

  np. wyrażenie regularne: `[a-zA-Z][a-zA-Z0-9]*`, nad alfabetem $A=\{a,b,\ldots,z,A,B,\ldots,Z,0,1,\ldots,9\}$

  można zastąpić ***definicją regularną*** postaci:

```
litera [a-zA-Z]
cyfra [0-9]
{litera}({litera}|{cyfra})*
```

  nad alfabetem $A = \{a,b,\ldots,z,A,B,\ldots,Z,0,1,\ldots,9\} \cup \{litera,cyfra\}$

- znaki specjalne:

```
. % \ / " * ( ) ? + | ^ $ [ ] - < > { }
```

- priorytety operatorów tworzenia wyrażeń regularnych w porządku malejącym:
  a) domknięcie zwrotne *****, domknięcie dodatnie **+**, opcjonalny wybór **?**, zbiór znaków **[]**, dopełnienie zbioru znaków **[^]**, konkatenacja wielokrotna **{}**
  b) konkatenacja
  c) alternatywa **|**
  d) prawy kontekst **/**, początek wiersza **^**, koniec wiersza **$**
  e) lewy kontekst **<>**
  f) koniec pliku **<<EOF>>**

- **przykłady** programów z różnymi typami wzorców:

%%	%%
`nie ;`	`"nie" ;`

Wejście: `Kto się nie nauczył, ten nie zda.`
Wyjście: `Kto się nauczył, ten zda.`

%%

```
^[a-zA-Z][a-zA-Z_0-9]*$  {printf("nazwa: %s", yytext);}
^[0-9]+(\.[0-9]+)?$      {printf("liczba: %s", yytext);}
^[^a-zA-Z0-9]+$          {printf("inne: %s", yytext);}
 .                       {;}
```

zastosowanie konkatenacji wielokrotnej:

```
%%
[a-zA-Z]{1,8}(\.[a-zA-Z0-9]{0,3})?      ECHO;
.                                          {;}
```

wykorzystanie znaków specjalnych:

```
%%
\ |\t|\n     {printf("biala spacja");}
.            {;}
```

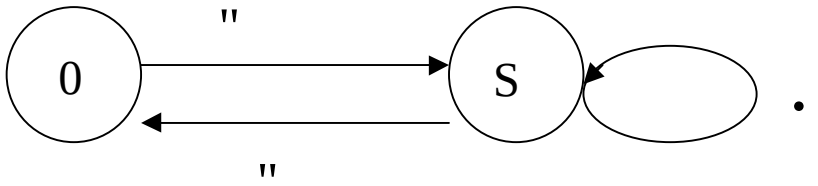
zastosowanie operatora lewego kontekstu:

`%start` *nazwa stanu* w części definicji pomocniczych (może być wiele)

<*nazwa stanu*> odwołanie do stanu we wzorcach reguł przetwarzania

np.:

```
%start string          definicja stanu S (pomijanie znaków łańcucha)
%%
<string>\"   {BEGIN 0;}
<string>.    ;
\"           {BEGIN string;} albo <0>\"     {BEGIN string;}
```



w stanie 0 następuje przepisywanie wejścia na wyjście, w stanie `string` (S) – pomijanie znaków

przykład zastosowania definicji regularnych i operatora prawego kontekstu:

wyrażenie regularne/podgląd

wyodrębnienie części pasującej do *wyrażenie regularne*,
o ile występuje po niej fragment pasujący do wyrażenia regularnego *podgląd*

np.:

```
numb [0-9]+                    nazwa wyrażenia
%%
{numb}                         {printf("Card ");}
{numb}"."{numb}?/[^\.]         {printf("Real ");}
".."                           {printf("Range ");}
"."                            {printf("Dot ");}
" "                            ;
```

dla wejścia:

```
1..5 37.5
```

otrzymamy na wyjściu:

```
Card Range Card Real
```

2.2.2 Zasady działania przetwornika (reguły dopasowywania wzorców)

- ***cykliczne analizowanie*** pliku wejściowego, którego ***przedrostki*** są ***dopasowywane*** do ***wzorców*** zawartych w regułach przetwarzania; wykonywanie akcji związanych z wzorcami
- reguła ***najdłuższego dopasowania*** (gdy przedrostek pasuje do kilku wzorców; bierze się pod uwagę także długość ciągu podglądanego, który jest z powrotem przesyłany na wejście)
- reguła ***pierwszego dopasowania*** (wybór wzorca występującego wcześniej, gdy ta sama długość dopasowania)
- ustawienie wartości zmiennych globalnych: `yytext` (dopasowany ciąg znaków, *przedrostek* pliku wejściowego, łącznie z częścią *podgląd* w sytuacji zastosowania prawego kontekstu), `yyleng` (długość *przedrostka* bez części dopasowanej do *podgląd*) i `yy_end` (długość całości dopasowanej, łącznie z częścią dopasowaną do *podgląd*), wykonanie akcji związanej z wzorcem dopasowania i usunięcie dopasowanego *przedrostka* z pliku wejściowego (część dopasowana do *podgląd* jest zwrotnie przekazywana na wejście)
- reguła ***zastępcza*** (przepisanie znaku wejściowego na wyjście), w wypadku braku dopasowania.

2.2.3 Akcje w regułach

- akcja ma postać instrukcji języka C (w tym akcja pusta `;` – pominięcie dopasowanego ciągu znaków albo kreska `|` – powtórzenie akcji z następnej reguły przetwarzania)
- w akcjach można wykorzystywać ***zmienne standardowe***, ***funkcje standardowe*** i ***makrowywołania***
- zestawienie ***makrowywołań***:

Makrowywołanie	Opis
`ECHO`	– `ECHO` – skopiowanie `yyleng` początkowych znaków zmiennej `yytext` do pliku wyjściowego
`BEGIN` *stan*	– `BEGIN` *stan* – zmiana bieżącego stanu początkowego przetwornika tekstu na stan *stan*
`REJECT`	– `REJECT` – przekazanie sterowania do kolejnej spośród aktualnie pasujących reguł przetwarzania
`yymore()`	– `yymore()` – modyfikacja standardowego algorytmu wyznaczania wartości `yytext`; w kolejnym kroku wartość zmiennej `yytext` powstaje poprzez sklejenie przedrostka ciągu wejściowego dopasowanego do ustalonego w tym kroku wzorca z bieżącą wartością zmiennej `yytext`
`yyterminate()`	– odpowiada instrukcji `return 0;`

`yyless(`*n*`)`	– `yyless(`*n*`)` – zwrotne przesłanie na początek pliku wejściowego wszystkich, z wyjątkiem *n* początkowych, znaków dopasowanych do aktualnego wzorca

- zestawienie ***funkcji standardowych***:

Funkcja	Opis
`input()`	– `input()`– wartością jest jeden wczytany, początkowy znak z nie-przetworzonej części pliku wejściowego
`unput(`*c*`)`	– `unput(`*c*`)` – dołącza wskazany znak *c* na początek nieprzetworzonej części pliku wejściowego
`yywrap()`	– `yywrap()` – obsługuje sytuację: – a) zakończeniu działania, w sytuacji napotkania końca aktualnie przetwarzanego pliku i braku dalszych plików do przetwarzania; – b) kontynuacji działania w odniesieniu do nowego pliku wejściowego, w sytuacji napotkania końca aktualnie przetwarzanego pliku wejściowego i wystąpienia żądania przetwarzania kolejnego pliku;
`yyerror(`*msg*`,...)`	– `yyerror(`*msg*`,...)` – wysyła do standardowego pliku z błędami komunikat o treści *msg*; po wysłaniu tego komunikatu przetwornik tekstu kontynuuje działanie na pozostałej części pliku wejściowego.

	– `yyerror(`*msg*`,...)` jest równocześnie funkcją standardową systemu YACC.

- zestawienie ***zmiennych standardowych***:

Zmienna	Opis
`yytext`	`char yytext[YYLMAX]` (ewentualnie `char *yytext`) – przechowuje przedrostek dopasowany do wzorca w bieżącej regule przetwarzania; jeśli wzorzec jest postaci *wregularne/podglad*, to w `yytext` znajdzie się sklejenie ciągów znaków dopasowanych do *rwregularne* i *podglad*; pojemność `YYLMAX` tablicy `yytext` jest ustalana za pomocą dyrektywy `#define`
`yy_end`	`int yy_end` – pamięta długość ciągu znaków aktualnie przechowywanego w `yytext;` jeśli `yytext` przechowuje przedrostek dopasowany do *wregularne/podglad*, to `yy_end` pamięta łączną długość tego przedrostka
`yyleng`	`int yyleng` – pamięta: a) długość całego ciągu znaków przechowywanego w `yytext`, jeśli jest nim przedrostek dopasowany do wyrażenia bez operatora prawego kontekstu `/` b) długość tego fragmentu ciągu znaków przechowywanego w `yytext`, który pasuje do *wregularne*, jeśli zmienna `yytext` przechowuje przedrostek dopasowany do wzorca z operatorem prawego kontekstu `/`, o postaci: *rwregularne/podglad*
`yylastc`	`char yylastc` – przechowuje ostatni znak przedrostka dopasowanego do wzorca poprzedniej

	aktywnej reguły przetwarzania; jeśli wzorzec ten był postaci *wregularne/podglad*, to jest to ostatni znak dopasowany do *wregularne*
`yyin`	`FILE *yyin` – wskazuje na aktualnie przetwarzany plik wejściowy; domyślnie jest nim `stdin`; redefinicja przyjmuje postać `yyin =` *plikwe*, gdzie *plikwe* to wskazanie na nowy plik wejściowy
`yyout`	`FILE *yyout` – wskazuje na plik wyjściowy, domyślnie na `stdout`; redefinicja za pomocą instrukcji `yyout =` *plikwy*, gdzie *plikwy* oznacza wskazanie na nowy plik wyjściowy
`yylineno`	`int yylineno` – przechowuje numer porządkowy aktualnie przetwarzanego wiersza pliku wejściowego
`yystart`	`int yystart` – przechowuje numer tego stanu automatu skończonego, w którym rozpoczyna się realizacja bieżącej reguły przetwarzania (automat skończony opisujący działanie przetwornika, jest budowany równolegle z opisem w języku C, standardowo umieszczanym w pliku `lex.yy.c`)

2.2.4 Sekcja definicji pomocniczych

- ***deklaracje*** `extern` zmiennych zewnętrznych programu
- ***definicje zmiennych statycznych***, widocznych w pewnym fragmencie lub całym programie (definicje zmiennych automatycznych, których zakres widoczności ma być ograniczony do pojedynczej akcji programu należy umieścić w treści tej akcji)
- ***definicje stanów początkowych*** wykorzystywanych we wzorcach z operatorem lewego kontekstu; definicje te mają postać:

  `%start` ***stan***

  gdzie ***stan*** oznacza nazwę stanu początkowego używanego w programie
- ***dyrektywy*** `#include` włączania innych plików do programu
- ***dyrektywy*** `#define`
- ***definicje nazw wyrażeń regularnych***, postaci:

  – ***nazwa znaczenie***

  – gdzie ***nazwa*** oznacza kod mnemoniczny wyrażenia regularnego, a ***znaczenie*** definiuje jego treść

2.2.5 Sekcja podprogramów pomocniczych

- zawierają opisy działań w postaci funkcji języka C
- mogą redefiniować funkcje standardowe lub definiować specyficzne działania niestandardowe
- **przykłady** programów:

```
%{                                  sekcja definicji pomocniczych
  int num_count = 0;
%}
%%                                  sekcja reguł przetwarzania
[0-9]+              {printf("+"); REJECT;}
[\+\-]?[0-9]+       {ECHO; ++num_count;}
[\ \t\n]            {ECHO;}
%%                                  sekcja podprogramów (redefinicja funkcji standardowej)
int yywrap(void)
{
 printf("FILE CONSISTS OF %d NUMBERS", num_count);
 return 1;}


%%
```

```
 a          |                    powtórzenie akcji z następnej reguły
 ab         |
 abc        |
 abcd       ECHO; REJECT;
 .|\n       ;

%{
int Top=0, Stos[100];     kalkulator wyrażeń w ONP
%}
%%
[0-9]+        {push1(atoi(yytext));}
"+"           {push1(pop()+pop());}
"*"           {push1(pop()*pop());}
[\ \t]        ;
\n            {printf("%d\n",pop());}
.             {yyterminate();}
%%
int push1(int x)
  {Stos[++Top]=x; return 1;}
int pop(void)
  {return Stos[Top--];}
```

dane (wyrażenia arytmetyczne w ONP) i wyniki (wartości wyrażeń):

```
2                           2
3 5 +                       8
2 8 *                       16
2 3 5 + *                   16
2 3 5 * +                   17
```

```
%{
#include <stdlib.h>
#include <string.h>
typedef union {char *txt; float fval; int  ival;} u_type;
u_type attrib;
%}
%%
\+                      {printf("+");}
\*                      {printf("*");}
\(                      {printf("(");}
\)                      {printf(")");}
[0-9]*\.[0-9]+          {attrib.fval = atof(yytext);ECHO;}
[0-9]+\.                {attrib.fval = atof(yytext);ECHO;}
[0-9]+                  {attrib.ival = atoi(yytext);ECHO;}
[A-Za-z][A-Za-z0-9_]*   {attrib.txt = strdup(yytext);ECHO;}
[\ \t\n]                 {;}
.                       {yyerror("Unexpected symbol");}
```

2.2.6 Uwagi o uruchamianiu generatora Lex i przetwornika tekstu

- jeśli plik z programem w języku Lex, opisującym działanie przetwornika tekstu nazywa się *przetw*`.l`, to polecenie:

  `lex -o` *przetw*`.c` *przetw*`.l`

  spowoduje utworzenie przetwornika (automatu) w języku C i umieszczenie go w pliku o nazwie *przetw*`.c`
- przetwornik należy skompilować, korzystając z dostępnego kompilatora języka C (np. `cc` lub `gcc`), wskazując plik biblioteczny systemu Lex o nazwie `l` (opcja ma postać `-ll`):

  `gcc` *przetw*`.c -ll`
- powstanie plik wykonywalny o standardowej nazwie `a.out`, który można następnie stosować do przetwarzania tekstów, np.:

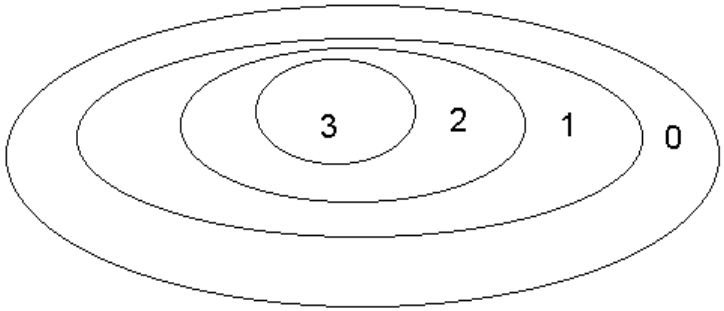
  `a.out < `*test.in*` > `*test.out*
  `a.out < `*test.in*
  `a.out > `*test.out*
  `a.out`

3. Elementy teorii języków formalnych

Metody definiowania składni języka <Σ, Syn>

- metody definiowania składni dzielimy na: ***generacyjne*** i ***akceptorowe***
- w metodach ***generacyjnych*** podaje się ***zbiór reguł generacji*** (formalny system generacji składni) poprawnych napisów nad podanym alfabetem (np. gramatyki kombinatoryczne N. Chomsky'ego)
- w metodach ***akceptorowych*** definiuje się nad alfabetem automat/akceptor (formalna definicja automatu), który potrafi zbadać przynależność napisu do języka (wszystkie napisy zaakceptowane przez automat definiują język)
- każdemu typowi gramatyki odpowiada typ automatu (podejścia są zatem komplementarne)
- ***translator*** jest przetwornikiem napisów, od akceptora różni się tym, że ma zdefiniowany alfabet wyjściowy i potrafi generować napisy w języku nad tym alfabetem (formalnie, jest automatem „z wyjściem").

3.2 Hierarchia języków wg N. Chomsky'ego:



– klasy języków uporządkowane relacją zawierania się zbiorów;

– języki klasy "**0**" nazywamy ***obliczalnymi***, są klasą najszerszą, matematycznie są przykładem zbiorów rekurencyjnie przeliczalnych (częściowo rozstrzygalnych);

– języki klasy "**1**" nazywamy ***monotonicznymi***, matematycznie są przykładem zbiorów rekurencyjnych (rozstrzygalnych);

– języki klasy "**2**" nazywamy ***bezkontekstowymi***; wyróżnia się istotne z praktycznego punktu widzenia podklasy **LL** i **LR** (generator YACC);

– języki klasy "**3**" nazywamy regularnymi; biorą swą nazwę od jednego z formalizmów ich definiowania zwanego ***wyrażeniami regularnymi***;

3.3 Formalny system generacji składni (podejście generacyjne)

F = < Σ, Syn, P, s>

Σ – alfabet

Syn – syntaktyka

P – zbiór reguł generacji napisów ze zbioru **Syn**

s – wyróżniony zbiór napisów początkowych;

- przykładem systemu generacji jest ***formalny system kombinatoryczny***:

  F = < Σ, Syn, P, s >, gdzie $\mathbf{P} \subseteq \Sigma^* \times \Sigma^*$, $\mathbf{s} \in \Sigma^*$;

- przykładem systemu kombinatorycznego jest ***gramatyka*** kombinatoryczna (***klasa 0***), zdefiniowana przez Noama Chomsky'ego:

$$\mathbf{G = < N, \Sigma, P, S >} \quad \mathbf{N \cap \Sigma = \varnothing}$$

N – skończony niepusty alfabet symboli pomocniczych (***nieterminalnych***),

Σ – skończony, niepusty alfabet definiowanego języka (symbole ***terminalne***),

$\mathbf{P} \subseteq (N \cup \Sigma)^+ \times (N \cup \Sigma)^*$, zbiór ***produkcji*** gramatyki (reguł generacji napisów),

$\mathbf{S} \in N$, wyróżniony symbol pomocniczy, zwany ***aksjomatem*** gramatyki;

- produkcję gramatyki, opisującą związek pomiędzy dwoma napisami (np. napisem α i napisem β), zapisujemy zwyczajowo $\alpha \rightarrow \beta$, gdzie strzałka oznacza ***znak zastąpienia***, a cały napis poleca zastąpić napis α napisem β;
- relacja ***wywodu bezpośredniego*** $\Rightarrow$ (oraz jej domknięcie $\Rightarrow^*$):

  $xabcy \Rightarrow xdvy$ o ile w zbiorze P istnieje produkcja $abc \rightarrow dv$;
- język J(G) generowany gramatyką G:

$$\mathbf{J(G) = \{x \mid S \Rightarrow^* x \wedge x \in \Sigma^*\}}$$

- ***zdanie*** (lub słowo) to napis zbudowany wyłącznie z symboli terminalnych, uzyskany za pomocą relacji wywodu $\Rightarrow^*$; napis zawierający symbole terminalne i nieterminalne, również uzyskany za pomocą relacji wywodu $\Rightarrow^*$, nazywamy ***formą zdaniową***

- ***operacje na językach*** (zbiorach słów J, J_1, J_2):

  suma: $J_1 \cup J_2 = \{x \mid x \in J_1 \vee x \in J_2\}$

  przekrój: $J_1 \cap J_2 = \{x \mid x \in J_1 \wedge x \in J_2\}$

  różnica: $J_1 - J_2 = \{x \mid x \in J_1 \wedge x \notin J_2\}$

  dopełnienie: $J' = \Sigma^* - J$

  złożenie: $J_1 J_2 = \{xy \mid x \in J_1 \wedge y \in J_2\}$

  potęgowanie: J^n, $J^0 = \{\varepsilon\}$, $J^k = JJ^{k-1}$

  domknięcie: $J^* = \bigcup_{i=0}^{\infty} J^i$

  domknięcie dodatnie: $J^+ = \bigcup_{i=1}^{\infty} J^i$, $J^+ = JJ^* = J^*J$, $J^* = J^+ \cup \{\varepsilon\}$

3.4 Formalna definicja automatu (podejście akceptorowe)

- w podejściu ***akceptorowym***: automat M ***akceptuje*** słowo (zdanie) s wtw. startując od konfiguracji początkowej ze słowem s na wejściu może wykonać skończony ciąg kroków kończących się konfiguracją końcową

- automat M definiuje język L(M), który jest zbiorem wszystkich słów s akceptowanych przez M

4. Translacja i kompilacja – podstawowe pojęcia

4.1 Zagadnienie translacji

tekst wejściowy ------->TRANSLACJA--------->*wynik translacji*

- zajmiemy się tylko wybranymi aspektami translacji: dotyczącymi tekstów napisanych w ***językach formalnych*** (sztucznych, np. językach programowania, dostępu do baz danych, publikacji informacji w Internecie – HTML, SGML, XML itd.) w modelu ***translacji sterowanej składnią*** (**L** = <Σ, **Syn** >)
- translacja sterowana składnią – proces, w którym po przeanalizowaniu struktury składniowej tekstu wejściowego jest on ***interpretowany*** lub na jego podstawie ***generowany*** jest ***tekst wyjściowy***
- ***narzędzia programistyczne*** realizujące translacje sterowane składnią to np.: AWK, ***Lex***, ***YACC***, sed, PERL i wiele innych, w tym języki programowania

– wyróżniamy ***dwie techniki*** translacji: ***interpretację*** i ***kompilację***

– proces kompilacji realizuje program zwany ***kompilatorem***:

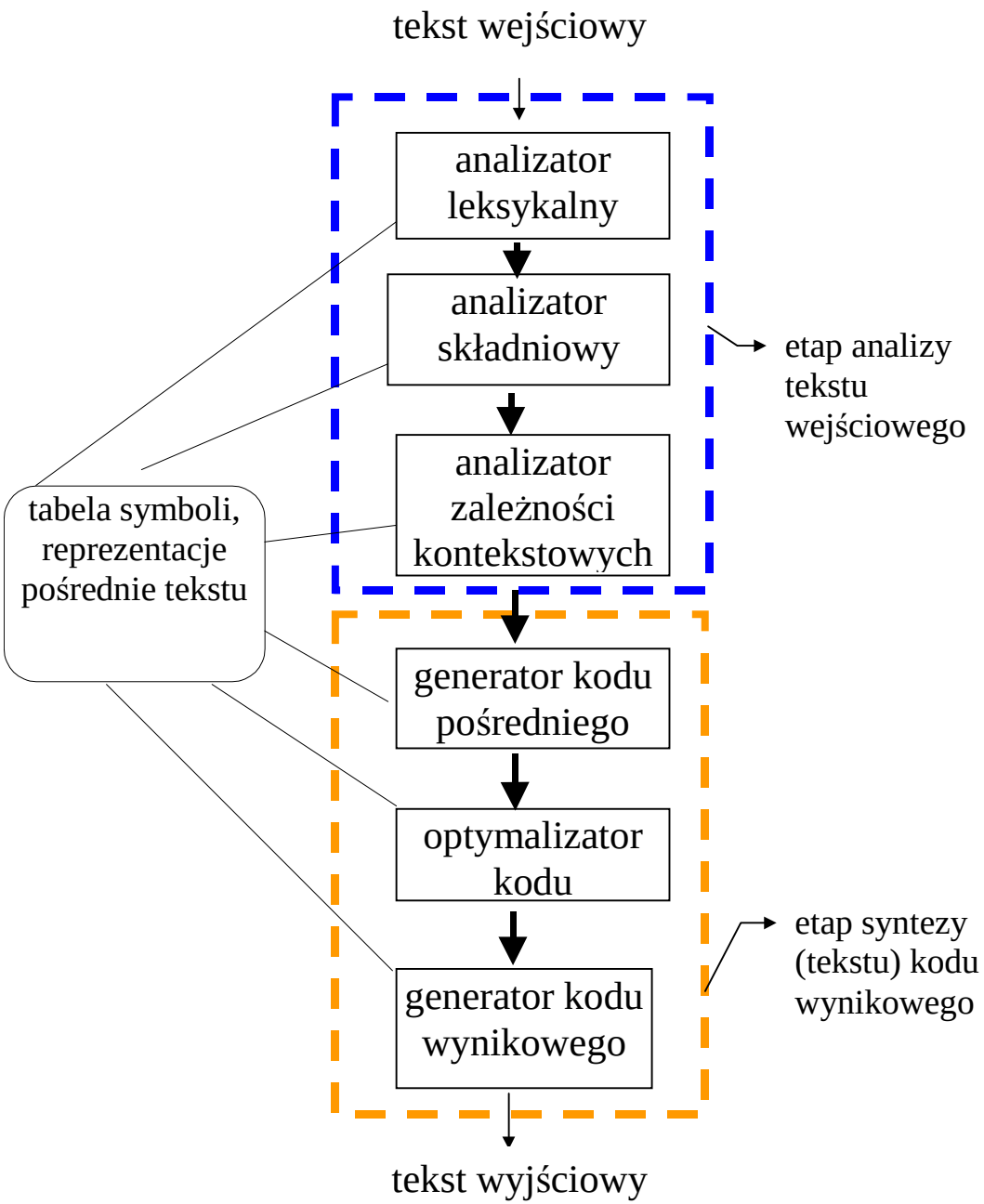
tekst wejściowy --> kompilator --> *tekst wyjściowy*

– kompilacja przebiega w dwóch ***etapach***:

a) ***analizy*** tekstu wejściowego i

b) ***syntezy*** tekstu wyjściowego,

które rozpadają się na ***fazy*** (każda faza przetwarza tekst wejściowy lub jego reprezentację):


tekst wejściowy
analizator leksykalny
analizator składniowy
analizator zależności kontekstowych
etap analizy tekstu wejściowego
tabela symboli, reprezentacje pośrednie tekstu
generator kodu pośredniego
optymalizator kodu
generator kodu wynikowego
etap syntezy (tekstu) kodu wynikowego
tekst wyjściowy

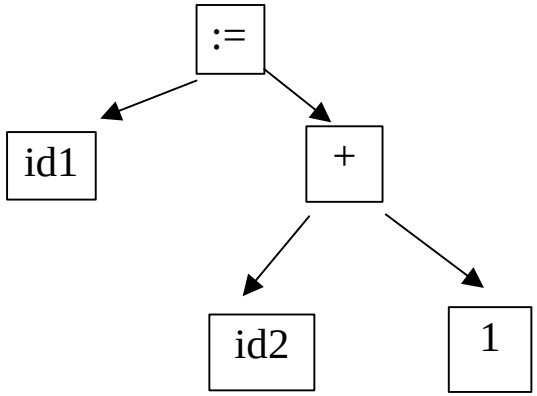
- kompilacja może być ***jedno-*** lub ***wieloprzebiegowa***
- przykład kompilacji:

```
program p1;
var x: real; i: integer;
begin
x:=i+1;
end.
```

Analizator leksykalny wyodrębni następujący zbiór ***jednostek leksykalnych***:

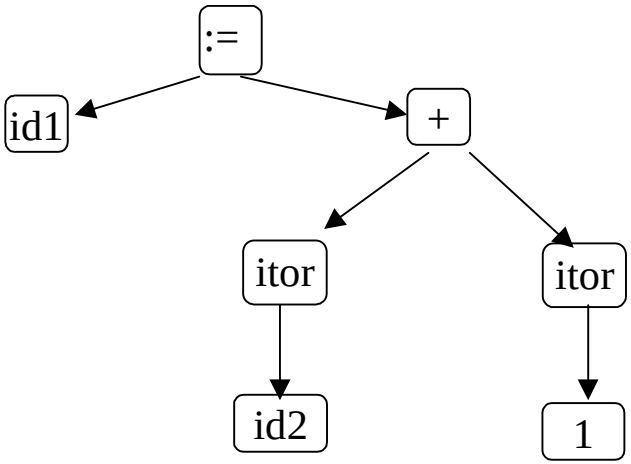
```
program p1 var x  : real ; i : integer begin := + 1 end .
```

Analizator składniowy utworzy ***drzewo rozbioru składniowego***, a informacje o identyfikatorach zmiennych umieści w ***tabeli symboli***:



Nazwa	Leksem	Typ	...
`id1`	`x`	`real`	`...`
`id2`	`i`	`integer`	`...`

Analizator zależności kontekstowych stwierdzi, że zmienne wykorzystane w instrukcji przypisania zostały zadeklarowane (informacje o tym zawarto w tabeli symboli) oraz zaznaczy w kodzie pośrednim, że należy przeprowadzić konwersję typu stałej **1** i zmiennej **`i`** z typu `integer` do typu `real` (operator ***itor***):



Generator kodu trójadresowego, wygeneruje dwie zmienne okresowe *temp1* i *temp2* i na podstawie powyższego drzewa utworzy zapis:

```
temp1 := itor(1)
temp2 := itor(id2)
id1 := temp1 + temp2
```

Optymalizator kodu dokona redukcji:

```
temp1 := itor(id2)
id1 := temp1 + 1.0
```

Na podstawie kodu pośredniego, ***generator kodu wynikowego*** utworzy zapis w kodzie asemblera, który następnie poddany zostanie ***asemblacji*** w celu uzyskania wersji wykonywalnej:

```
MOV id2, R1
ADD R1, #1.0
MOV R1, id1
```

4.2 Składnia Prostego Języka Programowania (PJP)

- notacja BNF (sposób zapisywania gramatyki języka)
- alfabet PJP (A)

```
<litera> ::= a|b|c|… z|A|B|C … Z

<cyfra> ::= 0|1|2|3|4|5|6|7|8|9

<operator jednoargumentowy> ::= +|-|nie

<operator dwuargumentowy> ::= +|-|/|*|:=|=|<|<=|>|>=|<>|i|lub

<słowo kluczowe> ::= program|poczatek|koniec|tablica|jesli|to|
                     przeciwnie|ilsej|dopoki|wykonuj|ikopod|
                     czytaj|drukuj|calkowite|rzeczywiste|boolowskie

A = <litera>∪<cyfra>∪<oparator jednoargumentowy>∪
    <operator dwuargumentowy>∪<słowo kluczowe>∪ {:;()[].}
```

- stałe

```
<stała> ::= <stała numeryczna>|<stała typu boolowskiego>
```

<stała numeryczna> ::= +<liczba>|-<liczba>|<liczba>

<liczba> (zdefiniować za pomocą wyrażenia regularnego, jako ćwiczenie)

<stała typu boolowskiego> ::= prawda|falsz

<identyfikator> ::= <litera>| <identyfikator><litera>|
<identyfikator><cyfra>

- zmienne

<zmienna>::= <identyfikator>|<identyfikator>[<stała>]

- wyrażenia

<wyrażenie>::= <stała>|<zmienna>|
(<wyrażenie><operator dwuargumentowy><wyrażenie>)|
(<operator jednoargumentowy><wyrażenie>)

- instrukcje

<instrukcje>::= <instrukcja>|
<instrukcja>; <instrukcje>

- instrukcja

```
<instrukcja>::= <zmienna>:= <wyrażenie>|
                jesli <wyrażenie> to <instrukcja> przeciwnie
                <instrukcja> ilsej|
                dopoki <wyrażenie> wykonuj <instrukcja> ikopod|
                poczatek <instrukcje> koniec|
                czytaj <zmienna>|
                drukuj <wyrażenie>
```

- typy

```
<typ>::= calkowite|rzeczywiste|boolowskie
```

- deklaracje

```
<deklaracje>::= <deklaracja>|
                <deklaracja>;<deklaracje>
```

- deklaracja

```
<deklaracja> ::= <zmienna>:<typ>|
                 <zmienna>:<typ> tablica [<stała>]
```

- programy

```
<program>::= program <identyfikator> <deklaracje>; <instrukcje>.
```