Metody Realizacji Języków Programowania

Marcin Benke

MIM UW

4 października 2010

Co to jest kompilator?

Program który tłumaczy programy w języku wyższego poziomu na kod maszynowy procesora (np 80x86, Sparc) lub maszyny wirtualnej (np. JVM).
Istnieje wiele podobnych klas problemów/programów, gdzie

- analizujemy dane wejściowe (zwykle tekst)
- tłumaczymy na inny “język”.

Co robi kompilator?

- Wczytuje program, zwykle jako tekst.
- Sprawdza poprawność i dokonuje *analizy* programu.
- Myśli chwilę.
- Generuje kod wynikowy *(synteza)*.

Części kompilatora realizujące analizę i syntezę określa się czasem nazwami *front-end* i *back-end*

Analiza

- analiza leksykalna — podział na leksemy (“słowa”);
- analiza składniowa — rozbiór struktury programu i jej reprezentacja w postaci drzewa;
- analiza semantyczna — powiązanie użycia identyfikatorów z odpowiednimi deklaracjami; kontrola typów.

Każda z faz analizy powinna dawać czytelne komunikaty o napotkanych błędach

Trudne, ale bardzo ważne.

Faza analizy jest niezależna od języka docelowego.

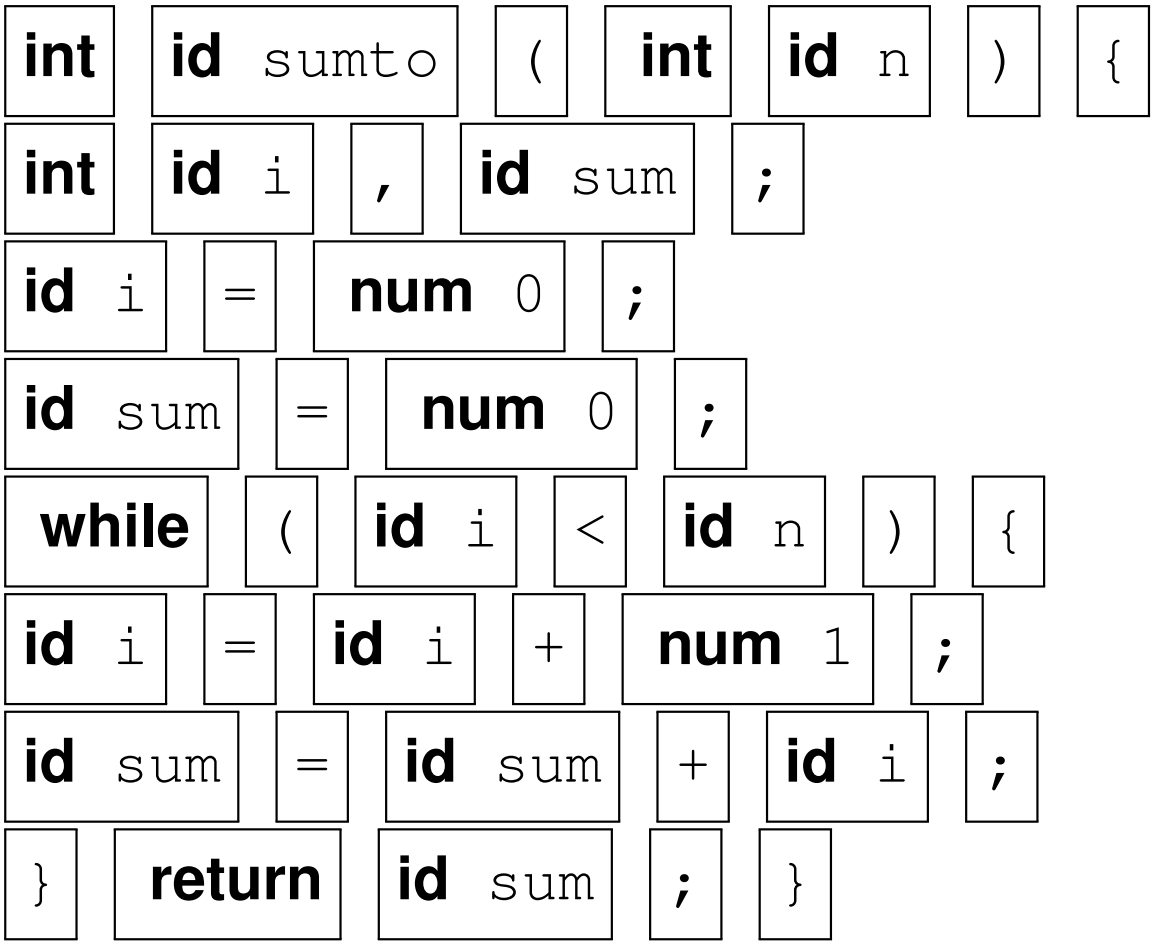
Przykład

Rozważmy prosty program w języku z rodziny C:

```
int sumto(int n)
{
  int i, sum;
  i = 0;
  sum = 0;
  while (i<n) {
    i = i+1;
    sum = sum+i;
 }
 return sum;
}
```

Analiza leksykalna

Dzielimy tekst na *leksemy*:



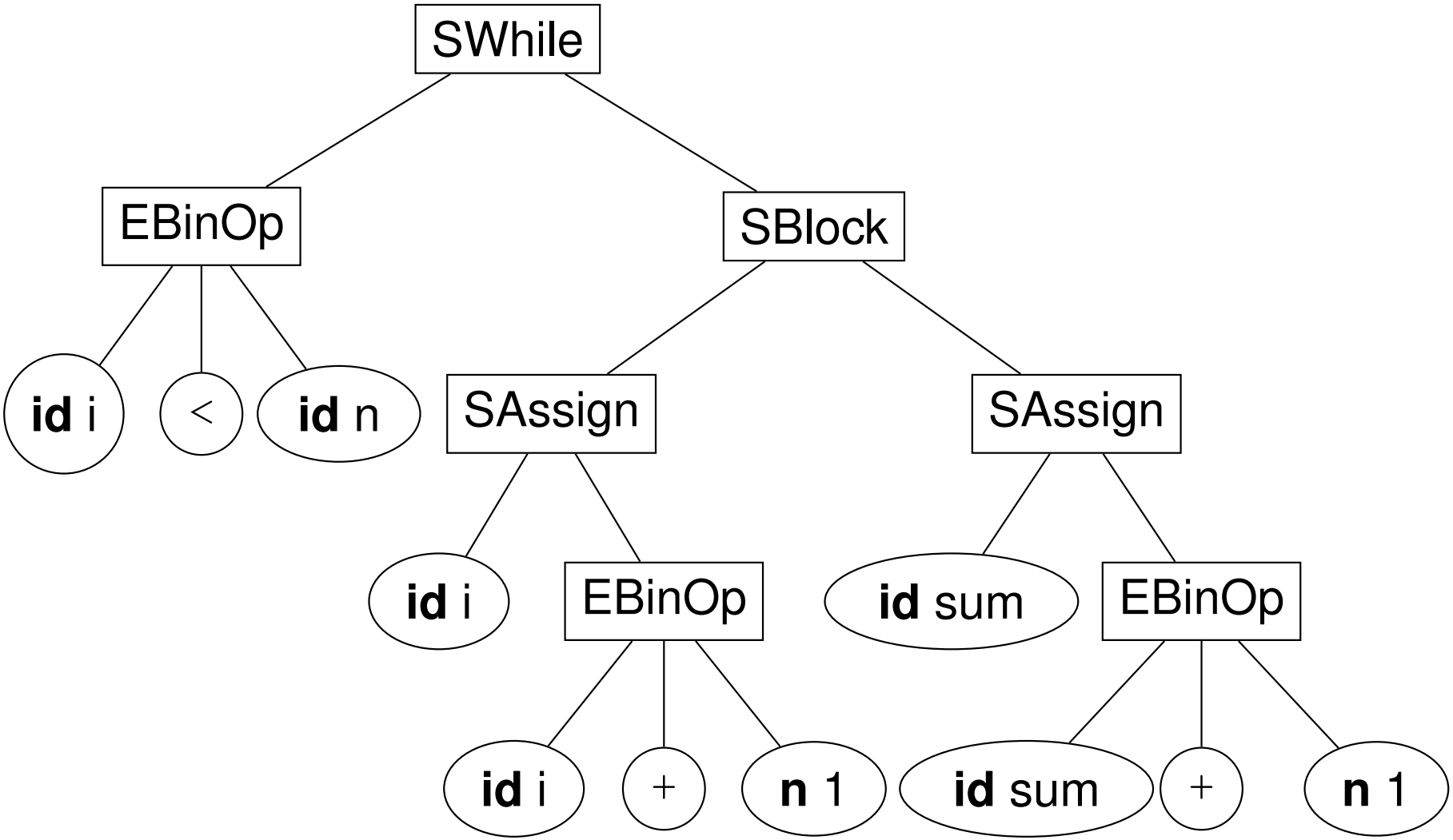
Analiza składniowa

Następnym krokiem jest kontrola poprawności składniowej programu.
Składnię języka opisujemy zwykle przy pomocy gramatyki, np:

$$
\begin{array}{rcl}
\mathit{Stmt} & ::= & \textbf{while}\ (\ \mathit{Exp}\ )\ \mathit{Stmt} \\
& | & \mathit{Var} = \mathit{Exp}\ ; \\
& | & \textbf{return}\ \mathit{Exp} \\
& | & \{\ \mathit{Stmts}\ \} \\
\ldots & &
\end{array}
$$

Analiza składniowa

Strumień leksemów zamieniamy na drzewo struktury:



Analiza semantyczna

- Analiza deklaracji
- Zapis informacji w *tablicy symboli*
- Kontrola poprawności użycia symboli i powiązanie z odpowiednimi deklaracjami (poprzez tablicę symboli).
- Kontrola (lub rekonstrukcja) typów.

Synteza

- Transformacja drzewa struktury do postaci dogodnej do dalszych przekształceń (kod pośredni)
- Planowanie struktur czasu wykonania (rekordy aktywacji, etc.)
- Ulepszanie kodu (“optymalizacja”)
- Wybór instrukcji
- Alokacja rejestrów
- Generacja kodu

Maszyna stosowa

- Argumenty i wyniki operacji na stosie

(+) Łatwo wygenerować kod z drzewa stuktury, np

```
genIntExp (EBinOp e1 op e2) = do
    genIntExp e1
    genIntExp e2
    intOp op

intOp "+" = emit "iadd"
```

(−) Trudno optymalizować

(−) Realne procesory nie są maszynami stosowymi

`sumto` dla maszyny stosowej (JVM)

```
.method public sumto()I
```

```
    iconst_0
    istore_2
    iconst_0
    istore_3
```

```
L1: iload_2
    iload_1
    if_icmpge L2
    iload_2
    iconst_1
    iadd
    istore_2
    iload_3
    iload_2
    iadd
    istore_3
    goto  L1
```

```
 L2: iload_3
     ireturn
.end method
```

`sumto` w asemblerze 80x86 (gas)

```
        .globl sumto
sumto:
        movl    4(%esp), %ecx ; ecx = n
        xorl    %eax, %eax    ; eax = 0
        testl   %ecx, %ecx    ; ecx <=0?
        jle     .L4           ; skok do L4
        xorl    %edx, %edx    ; edx = 0
.L5:
        addl    $1,%edx       ; edx += 1
        addl    %edx, %eax    ; eax += eax
        cmpl    %edx, %ecx    ; edx != n?
        jne     .L5           ; skok do L5
.L4:
        ret                   ; powrót
```

Plan wykładu

- Analiza leksykalna (dziś)
- Analiza syntaktyczna (4 wykłady)
- Analiza semantyczna (2 wykłady)
- Kolokwium
- Środowisko czasu wykonania
- Generacja kodu
- Optymalizacja (2 wykłady)
- Obsługa wyjątków
- Zarządzanie pamięcią
- Kompilacja języków funkcyjnych

Materiały

- `http://moodle.mimuw.edu.pl`
- `http://www.mimuw.edu.pl/~ben/Zajecia/Mrj`
- `http://wazniak.mimuw.edu.pl`
- A.V. Aho, M.S. Lam, R. Sethi, J.D. Ullman, Compilers: Principles, Techniques, and Tools, 2/E (w języku polskim dostępne jest tłumaczenie poprzedniego wydania: Kompilatory. Reguły, metody i narzędzia, Wydawnictwa Naukowo-Techniczne, Warszawa 2002).
- Materiały do wykładu pojawiają się sukcesywnie na **moodle.mimuw.edu.pl**
- Ostateczna wersja notatek **po wykładzie**.
- Kontakt ze mną: **ben@mimuw.edu.pl**
- Konsultacje czwartki 1400-1530 pokój 5750, prosze się umawiać.

Sprawy organizacyjne

Wykład+ćwiczenia
Kolokwium pod koniec listopada na wykładzie (kolokwium poprawkowe w styczniu,prawdopodobnie w sobotę).
Laboratorium: piszemy kompilator dla prostego języka

- Na ocenę **dst** — front-end + back-end dla VM.
- Na wyższą ocenę — rozszerzenia języka, optymalizacje lub generacja kodu dla konkretnego procesora.
- Wiele możliwości, szczegóły później.

Obecność (i aktywność!) na zajęciach jest wskazana. Prowadzący mają prawo (a nawet obowiązek) skreślić osoby, które opuszczą bez usprawiedliwienia zbyt wiele zajęć.

Proszę zadawać pytania!

Sondaż

- Czy jest ktoś, kto nie chodził na JPP?
- Jaki jest preferowany język programowania:
  - C/C++
  - Java
  - Haskell
  - *ML
  - Python
  - Lisp/Scheme/Clojure
  - inny

Analiza leksykalna

- Rodzaje leksemów
- Struktura leksykalna języka
- Analizatory leksykalne
- Generatory analizatorów leksykalnych
  Flex, Alex, Jlex,. . .

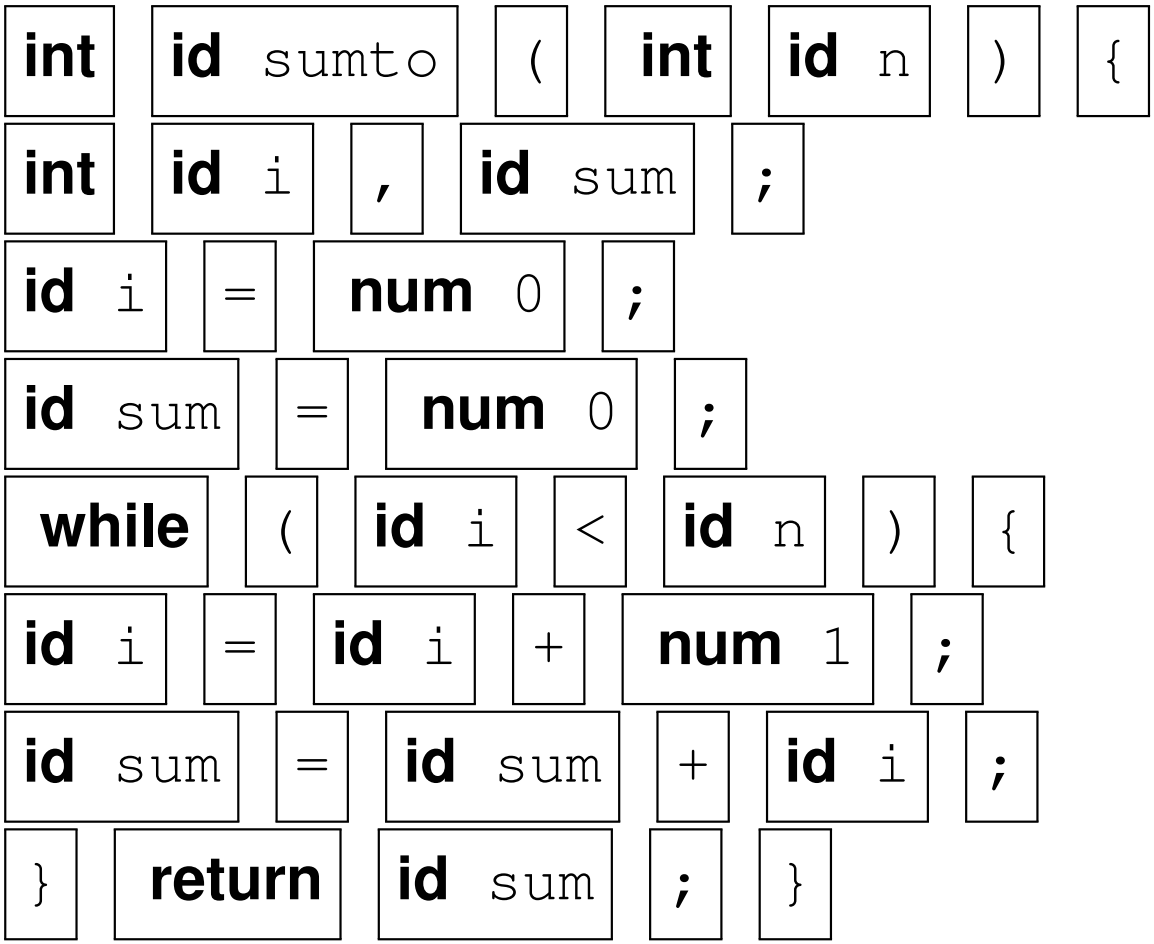
Ta część kompilatora nazywana jest *analizatorem leksykalnym*, lub w skrócie *lekserem*

Podstawowe rodzaje leksemów

- Słowa kluczowe, np. `for`, `do`
- Identyfikatory, np. `Foo`, `foo`
- Operatory, np. `++`, `>>=`
- Literały, np `3.14`, `'\n'`, `u"Gżegżółka"`
- “znaki przestankowe”, np. `{`, `;`

Analiza leksykalna

Dzielimy tekst na *leksemy*:



Leksemy

- Niektóre leksemy mają wartość semantyczną, np. **id** i, **num** 1
- Można opisać przy pomocy wyrażeń regularnych, np.

```
identyfikator = [a-zA-Z][a-zA-Z0-9]*
```

- Można rozpoznać przy pomocy DFA.

Reguła najdłuższego dopasowania

Zwykle wyrażenia regularne opisujące leksemy pozwalają na kilka różnych podziałów tekstu na leksemy, np.

- `--a` $\mapsto$ [–] [–] [**id** a] lub [--] [**id** a]
- `j23` $\mapsto$ [**id** j] [**num** 23] lub [**id** j23]

Zawsze wybieramy *najdłuższe możliwe dopasowanie*. Drugi przykład pokazuje wyraźnie dlaczego.

Odstępy

Znaki takie jak spacja, TAB, koniec linii zwykle są pomijane, mogą jednak służyć do rozdzielania leksemów, np.
`a++3` **vs** `a + +3`
Leksem może przechowywać swoją pozycję (wiersz, kolumna) w pliku źródłowym, np. dla celów raportowania błędów.
W niektórych językach pozycje niektórych leksemów mogą wpływać na rozbiór i znaczenie programu (np. Python, Haskell)

Układ (Python)

Przykład (Python)

```
for a, v in attrs:
    f(a,v)
    self.output(' %s="%s"' %(a,v))

self.output('>')
```

W tym przykładzie wcięcie determinuje zasięg pętli.

Układ (Haskell)

Znajdź szczegół różniący poniższe programy:

```
len xs = case xs of
           [] -> 0
           _:ys -> n+1
            where n = len ys
```

```
len xs = case xs of
           [] -> 0
           _:ys -> n+1
           where n = len ys
```

Układ (Haskell)

Pierwszy program odpowiada ciągowi leksemów takiemu jak:

```
len xs = case xs of {
           [] -> 0 ;
           _:ys -> n+1 where {n = len ys}
         }
```

Drugi zaś

```
len xs = case xs of {[] -> 0 ; _:ys -> n+1 }
         where {n = len ys}
```

(i jest niepoprawny)

Komentarze

- Przeważnie są pomijane (traktowane jako odstęp).
- Czasem mogą zawierać wskazówki dla kompilatora (pragmy), np

```
{-# OPTIONS -fglasgow-exts #-}
```

- Dwa rodzaje: do końca linii i “nawiasowe”.
- Te drugie mogą być zagnieżdżane:

```
/* Socket* foo(Addr& a) {
   /* TcpSocket *s; */
   Socket s = new UdpSocket();
   s->connect(a);
   return s;
} */
```

Język zagnieżdżonych komentarzy może nie być regularny, zwykle jednak jest rozpoznawany przez lekser przy użyciu dodatkowego licznika.

Lekser mozna napisać “ręcznie”...

...wystarczy zakodować wszystkie przypadki:

```
lexToken "" = (EOT,"")
lexToken s@(c:cs) = case c of
 c | isSpace c -> lexToken cs
   | isDigit c -> case lexInt s of
      (IntTok i,'.':sfrac) -> lexFloat s
      x -> x
   | isLower c -> let(vid, rest) = span isIdentChar s
   in case lookup vid keywords of
      Just keyword -> (keyword,rest)
      Nothing      -> (VarId vid, rest)
   | otherwise -> (ErrTok $ msg, rest) where
      msg = "Unexpected '"++[c]++"' in "++word
      (word,rest) = span (not.isSpace) s
...
```

Generatory analizatorów leksykalnych

- Pisanie analizatora leksykalnego jest zwykle żmudne.
- Dlatego przeważnie jest on generowany automatycznie przez narzędzia takie jak Flex (C,C++), JLex (Java), Alex (Haskell), Ocamllex, C#Lex,...
- Narzędzia takie generują program realizujący automat rozpoznający leksemy na podstawie opisu złożonego z wyrażeń regularnych i przypisanych im akcji.

Ogólna postać definicji leksera

definicje
%%
reguły (wzorce+akcje)
%%
funkcje pomocnicze (w C)

W sekcji definicji mogą się pojawić definicje klas znaków, np.

```
CYFRA       [0-9]
LITERA      [A-Za-z]
```

a także deklaracje opcji Flexa oraz używanych dalej zmiennych i funkcji (te ostatnie muszą być wcięte lub ujęte między znaczniki `%{` i `%}`.

Wzorce

`x`	znak `x`
`.`	dowolny znak (oprócz `\n`)
`[xyz]`	dowolny znak spośród `x,y,z`
`[abj-o]`	klasa znaków: `a,b`, od `j` do `o`
`[^A-Z]`	dopełnienie klasy znaków
r`*`	zero lub więcej *r*
r`+`	jedno lub więcej *r*
r`?`	zero lub jedno *r*
r`{2,5}`	*r* od 2 do 5 razy
`{4}`	*r* dokładnie 4 razy
rs	konkatenacja *r* i *s*
r`|`*s*	*r* lub *s*
r`/`*s*	*r* w kontekście *s*
`^`*r*	*r* na początku linii
r`$`	*r* na końcu linii

Wywoływanie leksera: `yylex()`

`lc.l`

```
int lines = 0, chars = 0 ;

%option noyywrap
%%
\n      ++lines ; ++ chars ;
.       ++chars ;
%%
void main()
{
yylex() ;
printf("Lines: %d, chars: %d\n", lines, chars) ;
}
```

Opcja `noyywrap` powoduje, że lekser nie wywołuje `yywrap()` przy napotkaniu EOF, ale zakłada, ze nie ma więcej plików do analizy.

Zawartość leksemu: zmienna `yytext`

```
%option main
int suma=0 ;
%%
[0-9]+          suma += atoi(yytext) ;
suma            printf("%d\n", suma) ;
\n
.                 /* nic */

> ./dodaj
11
22
33
suma
66
```

Klasy znaków i wspólne akcje

```
%option main
CYFRA    [0-9]
         double suma=0 ;
%%
{CYFRA}+(\.{CYFRA}+)?   suma += atof(yytext) ;
^\n                     |
suma                    printf("%g\n", suma) ;
\n
.                       /* nic */
```

Zawartość semantyczna leksemu: `yylval`

```
%{
#include "exp.tab.h"
%}
%option noyywrap
%%
[[:digit:]]+  {
          yylval = atoi( yytext ) ;
          return (int)NUM;
        }
.   { return (int)(yytext[0]); }
\n  { return (int)'\n'; }
%%
```

Nazwa `yylval` nie ma znacznenia dla Flexa, jest tylko użyteczną konwencja, ale inne narzędzia (np. Bison) mogą na niej polegać.

Komentarze

```
...
%start COMMENT
%%
<INITIAL>[0-9]+   {return numtoken(atoi((yytext));}
...
<INITIAL>"/*"     {depth=1;yybegin(COMMENT);}
<COMMENT>"/*"     {depth++;}
<COMMENT>"*/"     {depth--;
                   if(!depth)
                     yybegin(INITIAL);
                  }
<COMMENT>.        {}
```

Flex — użycie

`Makefile`

```
explex.c: exp2.l
        flex -o $@ $<
```

Jeśli pominąć `-o`, Flex stworzy plik `lex.yy.c` (dla kompatybilności ze starszym narzędziem Lex).

Alex

Generator lekserów dla Haskella; ta sama idea, trochę inna składnia.

```
{
module CalcLex(alexScanTokens) where
import CalcToken(Token(..))
}
%wrapper "basic"
$digit = 0-9
$alpha = [a-zA-Z]
tokens :-
  $white+                                ; -- whitespace
  "--".*                                 ; -- comment
  let                                    { \s -> Let }
  in                                     { \s -> In }
  $digit+                                {\s -> Int (read s)}
  [\=\+\-\*\/\(\)]                       {\s -> Sym (head s)}
  $alpha [$alpha $digit \_ \']*          { \s -> Var s }
```

Alex — użycie

```
ben@sowa$ alex CalcLex.x
ben@sowa$ ghci CalcLex.hs
GHCi, version 6.12.1...
Ok, modules loaded: CalcLex, CalcTokens.
*CalcLex> alexScanTokens "let x = 1 in x +x"
Loading package array-0.3.0.0 ... linking ... done.
[Let,Var "x",Sym '=',Int 1,In,Var "x",Sym '+',Var "x"]
```

Dla ciekawych: yywrap...

```
<<EOF>> {
 if ( --include_stack_ptr < 0 ) {
   yyterminate() ; // Really end
 } else {
    lineno = lineno_stack[include_ptr] ;
    strcpy( the_file_name,
            file_name_stack[include_ptr]
          ) ;
    yy_switch_to_buffer(include_stack[include_ptr]);
    return (int) ';' ;
  }/* if */
 } /* <<EOF>> */
%%
int yywrap() {
  return 1; /* there may be more input */
}
```