تحلیلگر لغوی، به عنوان ورودی، متن برنامه مورد کامپایل را **کاراکتر به کاراکتر** خوانده و سپس اجزاء زبان را در قالب توکن مشخص کرده و در اختیار **تحلیلگر نحوی** قرار می دهد.

ارسال **توکن** زمانی صورت می گیرد که تحلیلگر نحوی **درخواست** کند.
به هنگام تشخیص توکن، تحلیلگر لغوی از نقطه جاری متن برنامه، کاراکترها را یکی یکی خوانده تا به یک **جداکننده** برسد.

این جداکننده ها صرفاً می توانند خاصیت جداکنندگی داشته باشند، مانند کاراکتر جای خالی، تب و یا اینتر؛ چنین کاراکترهایی توسط تحلیلگر لغوی کنار گذاشته می شوند و ارزشی برای پارسر یا سایر واحدها ندارند.

اما برخی دیگر از جداکننده ها، مانند **سمیکالن، علامت مساوی، به اضافه** و ... خود به عنوان یک توکن نیز شناخته می شوند.

رابطه بین تحلیلگر نحوی و لغوی

ساختار توکن ها

در یک توکن معمولاً اطلاعات زیر وجود دارد:

- ✓ **شماره سطر و ستون، شماره بلاک، شماره ترتیب بلاک در برگیرنده، نوع لغت،خود لغت**

شماره سطر و ستون به هنگام تشخیص خطا کاربرد دارد.

Blockno شماره بلاکی است که در آن یک لغت شناسایی شده است.

Blockord هنگامی که بتوان بلاک های تودرتو تعریف کرد مورد استفاده قرار می گیرد.

```
struct token{
   int row,col;
   int blockno;
   int blockord;
   enum symbolType;
   char Name[40];}
```

در قطعه کد مقابل ۳ بلاک وجود دارد.

```
void f()
{ int i=1;
   printf("%d" , i);
}

void g()
{ int i=2;
   { int i=3;
      printf ("%d" , i);
   }
 printf ("%d" , i);
}
```

Block No = 1
Block Ord = 0

Block No = 2
Block Ord = 0

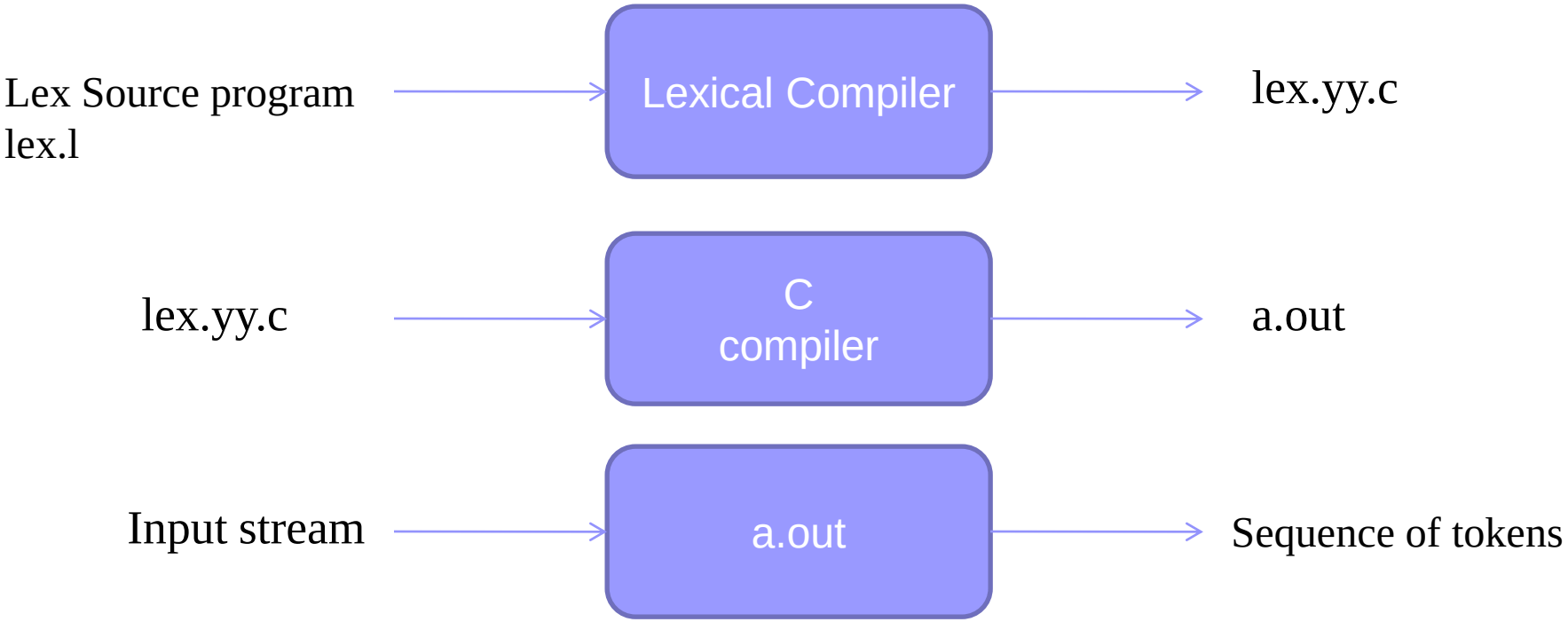
Block No = 3
Block Ord = 2

در مثال زیر، j در بلاک دوم تعریف نشده اما مورد استفاده قرار گرفته است. در چنین مواردی ترتیب بلاک در فازهای بعد مفید است:

```
{ int i=1,j=2;
   { int i=2;

printf("%d%d",i,j);
    }
  printf("%d",i);
}
```

Lexical Analyzer Generator – Lex

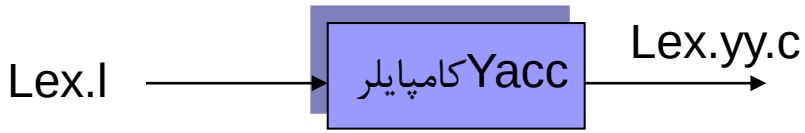
Lex+ yacc



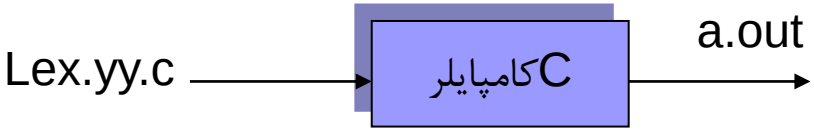
تولیدکننده تحلیلگر لغوی - Lex

روش ایجاد تحلیلگر توسط Lex

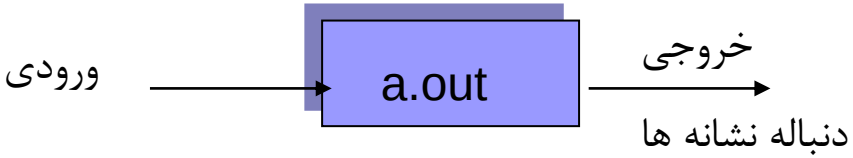
۱- آماده شدن پرونده ای حاوی مشخصه تحلیلگر برای Lex



۲- تبدیل محتوای پرونده به برنامه در زبان C



۳- کامپایل برنامه تولیدی همراه کتابخانه برنامه تحلیل لغوی



۴- خروجی: برنامه تحلیلگر لغوی

Lex -اجزای برنامه

بخش های برنامه مبدا Lex

- تعاریف
- قواعد ترجمه
- توابع

ترتیب در متن برنامه مبدا Lex

declarations
%%
translation rules
%%
auxiliary functions

→ **Pattern {Action}**

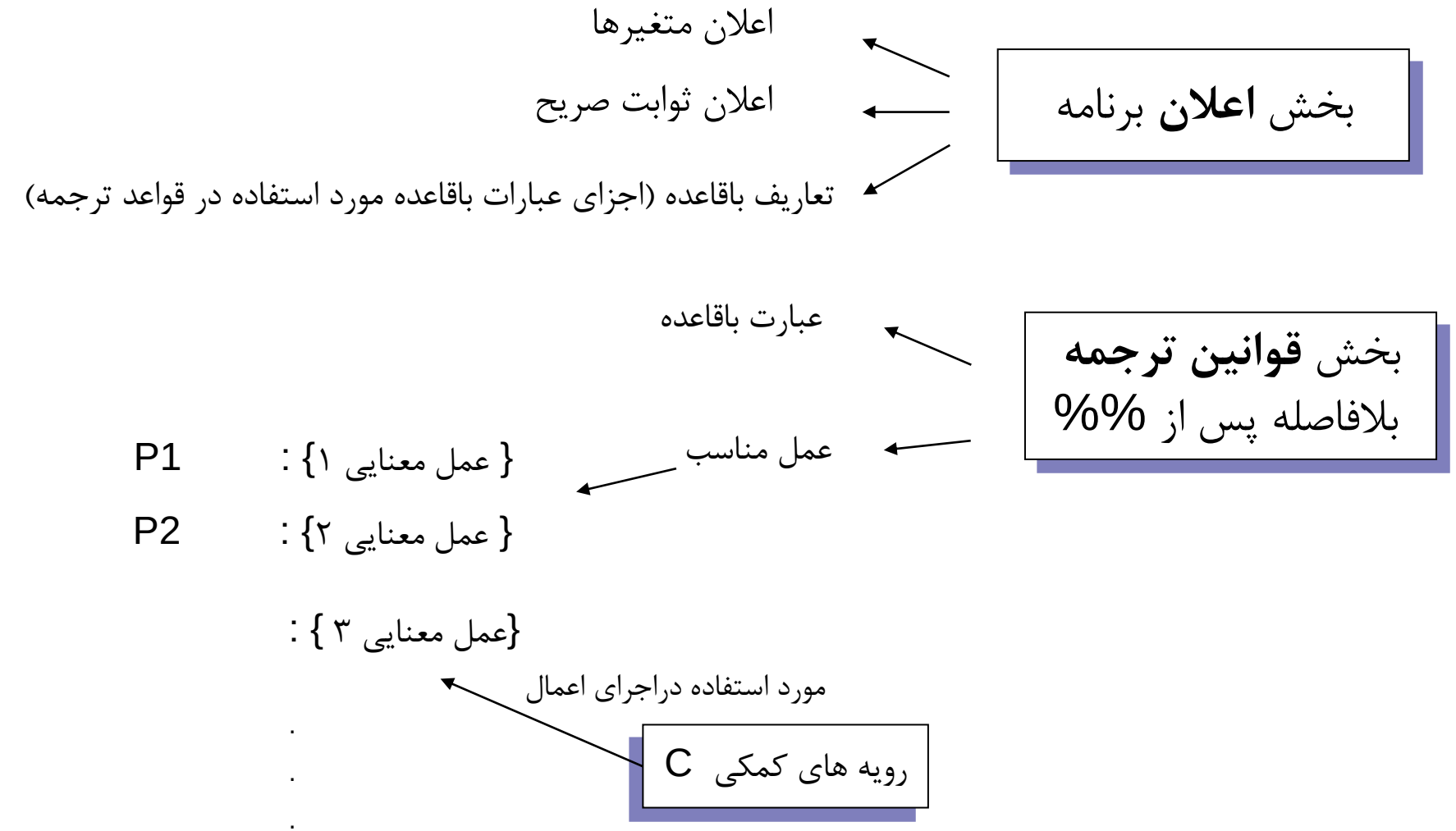
- تعاریف
- %%
- قواعد ترجمه
- %%
- توابع

Example

```
%{
    /* definitions of manifest constants
    LT, LE, EQ, NE, GT, GE,
    IF, THEN, ELSE, ID, NUMBER, RELOP */
%}

/* regular definitions
delim       [ \t\n]
ws          {delim}+
letter      [A-Za-z]
digit[0-9]
id          {letter}({letter}|{digit})*
number      {digit}+(\.{digit}+)?(E[+-]?{digit}+)?

%%
{ws}        {/* no action and no return */}
if          {return(IF);}
then        {return(THEN);}
else        {return(ELSE);}
{id}        {yylval = (int) installID(); return(ID); }
{number}    {yylval = (int) installNum(); return(NUMBER);}
…
```

```
Int installID() {/* funtion to install the
    lexeme, whose first character is
    pointed to by yytext, and whose
    length is yyleng, into the symbol
    table and return a pointer thereto
    */
}

Int installNum() { /* similar to
    installID, but puts numerical
    constants into a separate table */
}
```

تولیدکننده تحلیلگر لغوی



فصل سوم : تحلیل نحوی

- گرامر
- اشتقاق و تجزیه
- تعریف نحوگرا
- درخت نحوی
- تجزیه بالا به پایین و پایین به بالا
- تحلیل گر نحوی و خطاهای نحوی
- گرامر و گرامر مستقل از متن
- تجزیه بالا به پایین و پایین به بالا
- تجزیه پیشگو

تعریف ریاضی گرامر

تعریف: **ابزار توصیف ساختار جمله و رشته‌های یک زبان**

- گرامر G بوسیله چهارتایی $G = (V , T , S , P)$ تعریف می شود که در آن:
  - ✓ V : مجموعه ای متناهی از اشیا به نام متغیرها
  - ✓ T : مجموعه ای متناهی از اشیا به نام سمبل های پایانی (ترمینال ها/پایانه ها)
  - ✓ $S \in V$: سمبل ویژه ای به نام متغیر شروع
  - ✓ P : مجموعه ای متناهی از قوانین تولید (نحوه تبدیل رشته ها به یکدیگر را مشخص می کند)
- یک قانون تولید می تواند هر جایی که قابل اعمال باشد، به تعداد نامحدود، استفاده شود.
- با استفاده از گرامرها می توان با بکاربردن قوانین تولید با ترتیبهای مختلف، رشته های متعددی تولید کرد. مجموعه این رشته های پایانی زبانی است که توسط گرامر مورد نظر تولید می شود.
- مثال:

$G = (\{S\} , \{a,b\} , S , P)$

$S \rightarrow aSb$

$S \rightarrow \lambda$

$S \Rightarrow \underbrace{aSb} \Rightarrow \underbrace{aaSbb} \Rightarrow \underbrace{aabb}$

فرم های جمله — جمله

زبان تولید شده توسط گرامر G $L(G) = \{ a^n b^n : n \geq 0 \}$

انواع گرامر

گرامرها را به چهار دسته تقسیم می شوند:

۱- منظم

گرامری که همه قواعد آن به صورت $A \to Bx \mid x$ یا $A \to xB \mid x$ باشد. ($A, B \in V$ و $x \in T^*$)

۲- مستقل از متن

گرامری که در سمت چپ کلیه قواعد آن، فقط یک متغیر باشد.

۳- حساس به متن

گرامری که تمامی قوانین آن به فرم $x \to y$ باشند که در آن x وy عضو $(V+T)^+$ باشند و $|x| \le |y|$.

۴- بدون محدودیت

هیچ شرط و محدودیتی برای قواعد تولید ندارد. تنها محدودیت این است که λ نباید در سمت چپ قواعد تولید باشد.

گرامر با قاعده

فرم قوانین گرامر باقاعده

$A \longrightarrow Bx \mid x$

$A \longrightarrow xB \mid x$

x عنصری از الفبا و پایانه

B و A غیر پایانه

ID → letter A | A

A → letter A | digit A | A | λ

Int-Num → 1A | 2A | … | 9A

A → 0A | 1A | 2A | …| 9A | λ

مثال چند گرامر با قاعده

G: $S \rightarrow abSA$

$A \rightarrow Aa$

G: $S \rightarrow bS|cS|aB$

$B \rightarrow aB|cS\ |bC$

$C \rightarrow aB\ \ |\ bS$

گرامرهای خطی از راست و خطی از چپ

- گرامر مفروض $G = (V , T , S , P)$ در صورتی **خطی از راست** نامیده می شود که تمامی قوانین آن به فرم $A \rightarrow xB$ یا $A \rightarrow x$ باشند، که در آن، $A , B \in V$ و $x \in T^*$ هستند.

- یک گرامر در صورتی **خطی از چپ** نامیده می شود که تمامی قوانین آن به فرم $A \rightarrow Bx$ یا $A \rightarrow x$ باشند.

- همچنین **گرامر منظم** گرامری است که خطی از راست یا خطی از چپ باشد.

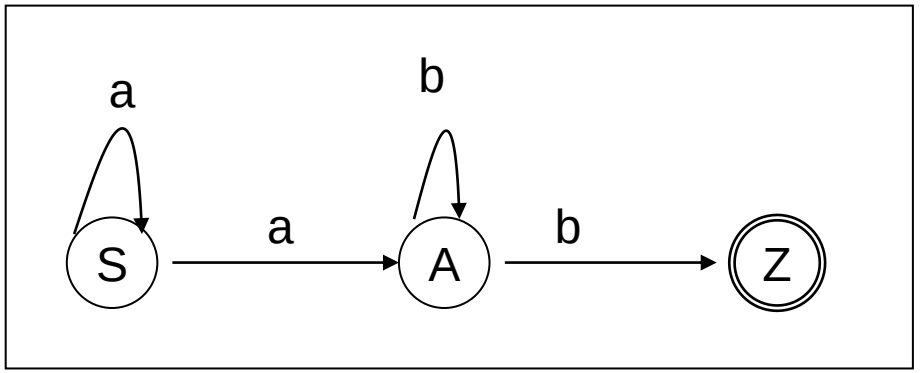
$S \rightarrow aS \mid Sb \mid \lambda$ این گرامر منظم نیست.

مثال از DFA

$S \rightarrow aS \mid aA$

$A \rightarrow bA \mid b$

تبدیل به
ماشین قطعی



اشتقاق	محاسبه	رشته پردازش شده
$S \rightarrow aS$	$[S , aabb] \rightarrow [S , abb]$	a
$\rightarrow aaA$	$[A,bb]$	aa
$\rightarrow aabA$	$[A , b]$	aab
$\rightarrow aabb$	$[Z, \ ]$	aabb

پذیرفته توسط ماشین

Context free grammars

Terminals پایانه

Nonterminals غیرپایانه

Start symbol نماد شروع

Productions مولدها، قوانین

expression -> expression + term

expression -> expression – term

expression -> term

term -> term * factor

term -> term / factor

term -> factor

factor -> (expression)

factor -> **id**

مثال از یک گرامر

$V = \{ E, F \}$

$T = \{ +, *, /, id \}$

$S = E$

$P = \{ E \rightarrow F * id \ , \ F \rightarrow F \mid E \ , \ F \rightarrow F + F \}$

رشته تولیدی نمونه

id * id+ id

استفاده از گرامرها

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid \mathbf{id}$

ورودی **X*(Y+Z)**

فواید گرامرها

۱- نمایش دقیق و قابل فهمی برای زبان

۲- امکان ایجاد پارسرهای کارآمد با قابلیت تشخیص ساختارهای نحوی درست و دقیق

۳- ایجاد ساختاری مناسب برای زبان جهت ترجمه صحیح و آشکارسازی خطا توسط گرامر درست طراحی شده

۴- سادگی اضافه نمودن ساختارهای جدید به زبان

عبارات باقاعده- دلایل استفاده برای لغوی زبان

۱- عدم نیاز به نمایش نوع قوی مانند گرامر برای توصیف قوانین ساده لغوی زبان

۲- امکان نمایش مختصرتر و قابل فهم تری برای نشانه ها نسبت به گرامر

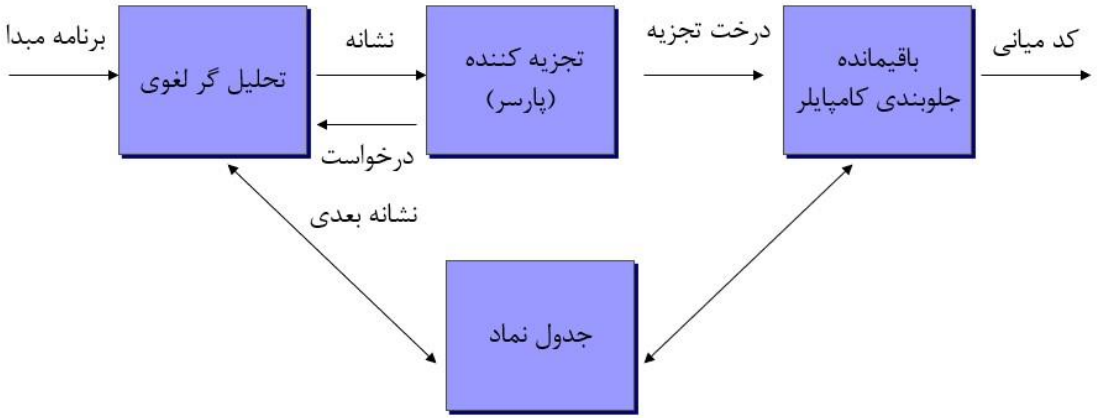
۳- ایجاد تحلیل‌گرهای لغوی کاراتر به صورت خودکار

۴- راه مناسبی برای پیمانه سازی جلوبندی کامپایلر به دو بخش قابل مدیریت با تقسیم ساختار زبان به لغوی و غیرلغوی

تجزیه کننده

دریافت رشته ای از نشانه ها از تحلیل گر لغوی و بررسی تعلق رشته به زبان توسط گرامر
انجام بررسی طبق ساختارهای نحوی زبان و هر مرحله گزارش خطاهای نحوی به اداره کننده خطا
رفع خطا برای پردازش ادامه ورودی بر اساس خطاهای متداول

تجزیه کننده– ارتباطات



موقعیت تجزیه کننده در مدل کامپایلر