



# به نام دانای نیآموخته

درس: کامپیالر

ترم: نیمسال اول (۰۵-۰۴)

دپارتمان: مهندسی کامپیووتر

مدرس: دکتر شیما شفیعی

## بارم بندی

۱. پایان ترم: ۶ نمره
۲. میان ترم: ۶ نمره
۳. حل تمرین: ۶ نمره
۴. پروژه گروهی: ۲ نمره

فعالیت مازاد:

مورد

فعالیت کلاسی

حضور در تمام جلسات

توسعه پروژه

نماینده کلاس

حل تمرین کلاس

## تاریخچه

توضیحات	بازه زمانی(میلادی)
کدنویسی با زبان اسembلی در سیستم‌های با حافظه کم و سرعت پایین	۱۹۵۰ تا ۱۹۶۰
استفاده از زبان‌های سطح بالا مانند فرتون (FORTRAN) برای افزایش قابلیت حمل، اعتماد و نگهداری برنامه‌ها	۱۹۶۰ دهه
نوشتن عملکردهای طولانی سیستم به صورت بخشی از سیستم عامل یا کتابخانه‌ها با زبان اسembلی	۱۹۷۰ دهه
پیشرفت بهینه‌سازی کامپایلرها، نوشتن بیشتر برنامه‌ها به زبان‌های سطح بالا بجز اجزای حیاتی	اواخر دهه ۱۹۷۰ و اوایل دهه ۱۹۸۰
عمل کردن کامپایلر بهتر از برنامه‌نویسان در استفاده از ثبات‌ها، پیچیده کردن کاری توسط پردازشگرهای برداری برای مدتی اندک	۱۹۸۰ دهه
تکامل کامپایلرها برای پردازشگرهای برداری، کاهش نیاز به دستورالعمل‌های پیچیده	اواخر دهه ۱۹۸۰ و دهه ۱۹۹۰
ظهور معماری RISC ، افزایش وابستگی به کامپایلرها، نوسان در عملکرد نسبت به معماری‌های CISC	دهه ۱۹۹۰ و بعد
شکل‌گیری پردازنده‌های فوق اسکالر، اجرای موازی دستورالعمل‌ها، پیچیدگی در آموزش برنامه‌نویسان	اواخر قرن بیستم

## اجزای یک سیستم برنامه نویسی

چهار بخش زیر در کنار هم فرایند تبدیل کد برنامه نویسی به برنامه قابل اجرا را مدیریت می کنند.

### الف) پیش‌پردازشگر (Preprocessor)

- قبل از فرایند کامپایل، روی کد منبع اعمال می شود.
- وظایفی مانند جایگزینی ماکروها، وارد کردن فایل‌های هدر، حذف کامنت‌ها و سایر اصلاحات متنی را انجام می‌دهد.
- خروجی پیش‌پردازش، کدی است که برای کامپایل آماده شده است.

### ب) کامپایلر (Compiler)

- کد منبع پیش‌پردازش شده را به زبان میانی یا زبان ماشین تبدیل می‌کند.
- شامل بخش‌های مختلفی مثل تجزیه (Parsing)، تحلیل معنایی (Semantic Analysis)، بهینه‌سازی و تولید کد است.
- هدف آن تولید کدی است که قابل اجرا روی پردازنده باشد.

## ادامه

### پ) اسembler (Assembler)

- کد زبان اسembلى (زبان سطح پايان) را به کد ماشين (صفر و يك) تبديل مى كند.
- کامپایلر در برخى زبانها يا معماريها ممکن است خروجى اسembلى توليد کند که اسembler آن را به زبان ماشين تبديل مى كند.

### ت) بارکننده و ویرايشگر الحق (Linker and Loader)

#### • بارکننده (Loader)

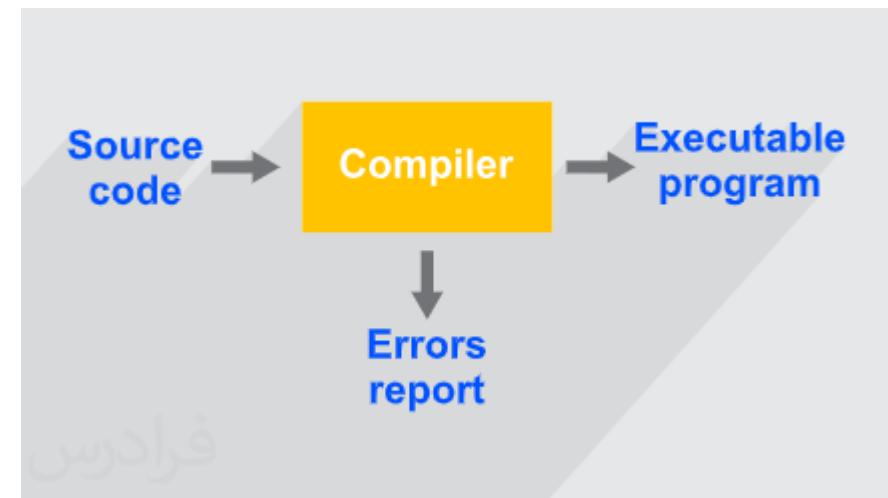
- برنامه های اجرایی را به حافظه اصلی منتقل می کند و آماده اجرای آنها می سازد.

#### • ویرايشگر الحق (Linker)

- فایل های شی مختلف که توسط کامپایلر یا اسembler تولید شده اند را به هم متصل می کند تا یک فایل اجرایی کامل ایجاد شود.
- کار کرد لینک کردن شامل اتصال کدهای کتابخانه ها، آدرس دهی صحیح و رفع ارجاعات بین ماژول ها است.

# کامپایلر

- کامپایلر نرم افزاری برای تبدیل کد منبع به کد شی است. به عبارت دیگر می‌توان گفت که کامپایلر کدهای نوشته شده به زبان سطح بالا (نزدیک به زبان انسان) توسط برنامه نویسان را به زبان دودویی ماشین تبدیل می‌کند.
- انجام این مرحله از اجرای برنامه‌ها و استفاده از کامپایلر به این دلیل الزامی است که کامپیوتراها تنها قادر به اجرای کدهای دودویی هستند و لذا کدهای سطح بالا باید به زبان ماشین ترجمه شوند.

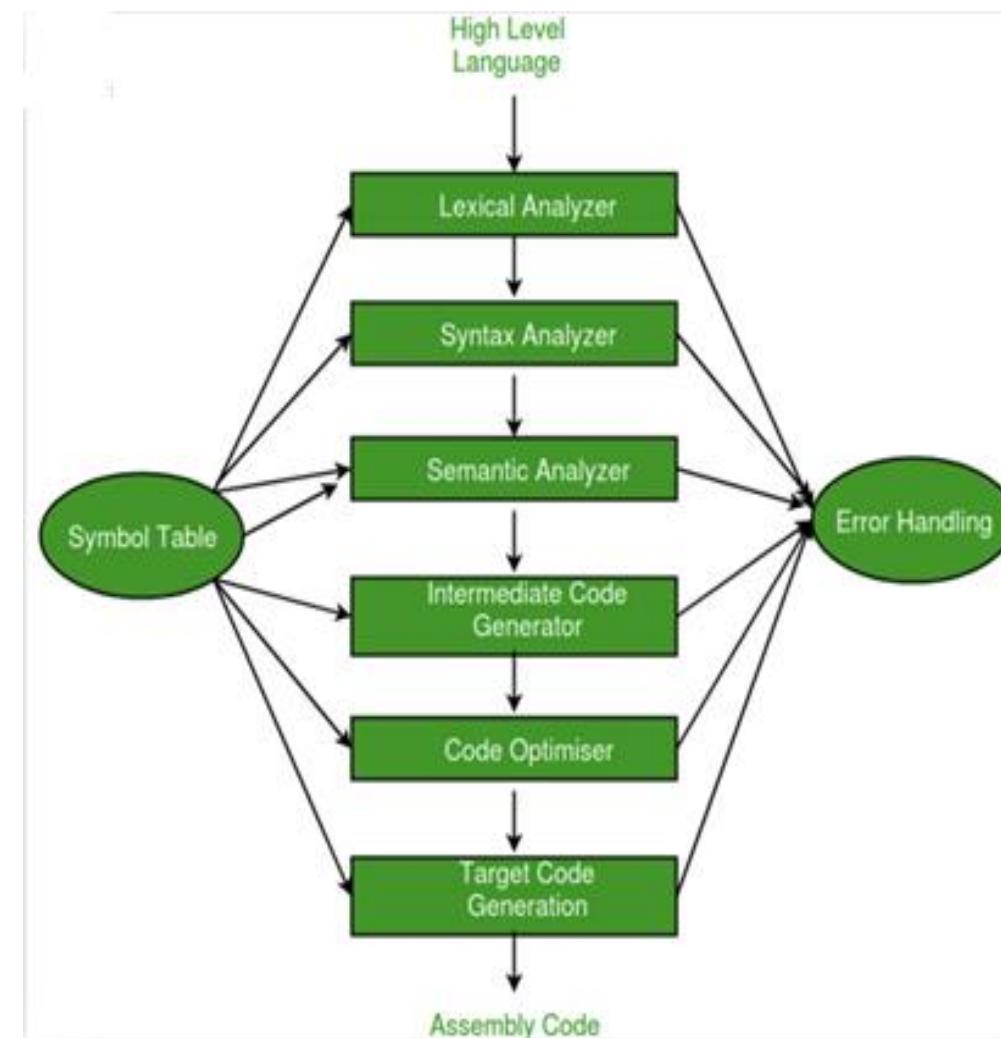


شكل ۱. فرآیند کامپایلر

## ادامه

- برخی از کامپایلرها زبان سطح بالا را در مرحله میانی نیز به زبان اسembلی تبدیل می‌کنند. در حالی که برخی از آن‌ها زبان سطح بالا را به طور مستقیم به کدهای دودویی ماشین تبدیل می‌کنند.
- کامپایل کردن همان فرآیند تبدیل کد منبع به کدهای دودویی ماشین است.
- تحلیل در کامپایلر به فاز اول ترجمه گفته می‌شود که در آن برنامه منبع به اجزای ساختاری کوچکتری مثل توکن‌ها و ساختارهای نحوی تقسیم می‌شود. این مرحله شامل تحلیل واژگانی، تحلیل نحوی و تحلیل معنایی است که هدف آن فهم ساختار و معنی برنامه است.
- سنتز در کامپایلر به فاز بعدی گفته می‌شود که در آن بر اساس نمایش میانی تولید شده در فاز تحلیل، کد هدف یا کد ماشین تولید می‌شود. در این مرحله بهینه‌سازی کد، تولید کد میانی و سپس تولید کد نهایی انجام می‌شود. هدف سنتز، تبدیل نمایش میانی به برنامه‌ای قابل اجرا در سخت‌افزار یا سیستم مقصد است.

## نمودار مراحل کامپایل



شكل ۲. نمودار مراحل کامپایل

# مراحل اجرایی در کامپایلر

## تحلیل‌گر لغوی (Lexical Analyzer) :

- کد منبع را به توکن‌های معنادار شکسته و تبدیل می‌کند (مثل کلمات کلیدی، شناسه‌ها، نمادها).
- توکن‌ها واحدهایی هستند که در مرحله بعدی پردازش می‌شوند.
- کار جداسازی متن خام از برنامه، حذف فاصله‌ها و کامنت‌ها را انجام می‌دهد.

## تحلیل‌گر نحوی (Syntax Analyzer) :

- توکن‌های ورودی را طبق قوانین دستور زبان (گرامر) بررسی می‌کند.
- ساختار درخت نحوی یا درخت تجزیه (Parse Tree) را می‌سازد.
- صحت ساختار دستوری برنامه را تایید می‌کند.

## تحلیل‌گر معنایی (Semantic Analyzer) :

- معنای دستورها و ارتباطات بین آن‌ها را مورد بررسی قرار می‌دهد.
- چک کردن تطابق نوع داده‌ها، بررسی اعلان متغیرها و بررسی نوع عملیات منطقی انجام می‌شود.
- در صورت کشف خطاهای معنایی، گزارش می‌دهد.

## ادامه

### :**(Code Optimizer)** بهینه‌ساز کد

- کد میانی تولید شده را بهینه می‌کند تا برنامه سریع‌تر و کم حجم‌تر اجرا شود.
- حذف کدهای اضافی، ترکیب دستورات و بهبود استفاده از منابع سخت‌افزاری انجام می‌شود.

### :**(Code Generator)** تولید کننده کد نهایی

- کد میانی به زبان ماشین یا زبان اسembلی مربوط به معماری مقصد تبدیل می‌شود.
- تخصیص ثبات‌ها و مدیریت آدرس‌دهی انجام می‌گیرد.

### :**(Error Manager)** مدیر جدول خطا

- مدیریت و ثبت خطاهای شناسایی شده در هر مرحله از کامپایل را انجام می‌دهد.
- امکان گزارش دقیق و دسته‌بندی انواع خطاها فراهم می‌کند.

### :**(Error Handler)** اداره کننده خطا

- مسئول تصمیم‌گیری در برخورد با خطاها، مثل متوقف کردن کامپایل، نمایش پیغام خطا یا تلاش برای ادامه فرایند کامپایل است.
- همچنین مکانیزم‌های رفع خطا و بازیابی اجرا را پیاده‌سازی می‌کند.

## تمرین ۱

فرض کنید در برنامه‌ای، یک متغیر عددی را به صورت رشته‌ای استفاده کرده‌اید. در کدام مرحله از کامپایل این خطأ شناسایی می‌شود؟ چرا؟

مهلت تحویل: ۶ مهرماه ۱۴۰۴

موفق باشید



دانشگاه شهید بهشتی کرمان

# به نام دانای نیآموخته

درس: کامپیویلر

ترم: نیمسال اول (۰۵-۰۶)

دپارتمان: مهندسی کامپیویلر

مدرس: دکتر شیما شفیعی

## اجزای یک کامپایلر

ساختار کلی یک کامپایلر که به دو بخش اصلی تقسیم شده است:

Front-end و Back-end بخش

Front-end : Lexical Analyzer, Syntax Analyzer, Semantic Analyzer

درک و بررسی صحت کد منبع

Back-end : Intermediate Code Generation, Optimization, Machine Code Generation

تولید کد قابل اجرا و بهینه سازی

اجزای هر زبان برنامه نویسی: Syntax و Semantics

## ادامه

### Syntax

قواعد و دستوراتی که نحوه نوشتن صحیح برنامه را تعیین می‌کند. اگر این قواعد رعایت نشود، خطای نحوی رخ می‌دهد.

مثال: در زبان برنامه‌نویسی Python ، جمله print("Hello") ساختار درست ندارد چون کوتیشن بسته نشده است.

### Semantics

مفهوم و هدف کدی است که نوشته شده، یعنی کاری که برنامه قرار است انجام دهد. گاهی کد ساختار درستی دارد ولی معنای آن اشتباه است یا منطقی نیست.

مثال: در Python ، کد print(5 / 0) ساختار درستی دارد اما از نظر معنایی اشتباه است چون تقسیم بر صفر تعریف نشده.

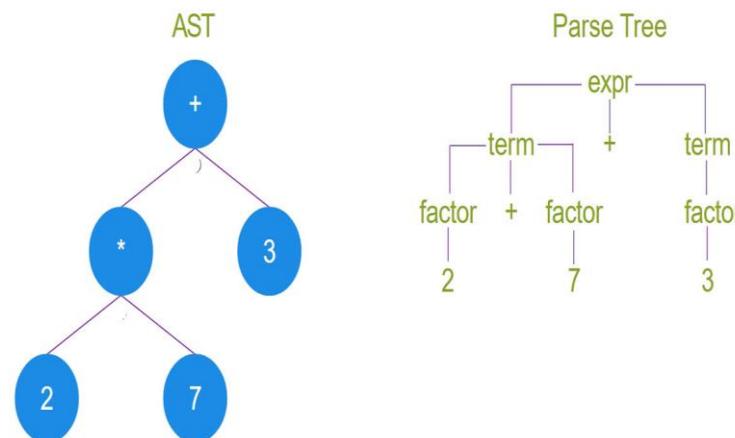
بنابراین، زبان برنامه‌نویسی باید هم از نظر ساختار درست باشد و هم از نظر معنا صحیح و منطقی باشد تا برنامه بتواند درست کار کند.

## اجزا

- کامپایلر با داشتن اطلاعات Syntax و Semantic زبان مبدا، کد منبع برنامه را به یک فرم نمایش تبدیل می‌کند که معمولاً به صورت درختی به نام درخت نحو انتزاعی یا (AST) (Abstract Syntax Tree) است.
- کامپایلر پس از Syntax Analysis که در آن ساختار کد بررسی می‌شود، خروجی را به صورت درخت نحو Parse Tree یا AST تولید می‌کند. این درخت ساختار سلسله‌مراتبی کد را نشان می‌دهد و عناصر زبان برنامه‌نویسی مانند عبارات، دستورات و عملیات‌ها را به صورت گره‌ها در خود دارد.
- سپس در مرحله Semantic Analysis، کامپایلر این درخت را برای بررسی تطابق معنایی کد با قوانین زبان (مانند تطابق نوع داده‌ها، بررسی تعریف متغیرها و غیره) بررسی می‌کند و اطلاعات معنایی لازم را به درخت اضافه می‌کند یا خطاهای معنایی را مشخص می‌کند.

## ادامه

EX:  $2 * 7 + 3$



- سمت راست:

این درخت کامل‌تر است و تمام جزئیات گرامری عبارت را نمایش می‌دهد.

- گره ریشه `expr` است که نشان‌دهنده کل عبارت است.

- گره‌ها شامل `term`, عملگرها (+ و \*), و `factor` (اعداد ۲، ۳ و ۷) هستند.

- نمایانگر کامل فرم نحوی و قواعد زبان برنامه‌نویسی است.

- سمت چپ :

این درخت ساده‌تر است و فقط ساختار معنایی و عملیات اصلی را نشان می‌دهد.

- دیگر بخش‌های غیرضروری مانند دسته‌بندی‌های `term` و `factor` حذف شده است.

- عملگرها و اعداد بطور مستقیم به عنوان گره به نمایش درآمده‌اند.

- درخت کوچک‌تر و برای تحلیل معنایی و بهینه‌سازی کاربرد دارد.

## مثال

**int 2x**

توكن های عبارت رو به رو به صورت زیر هستند:

.<Token,Lexeme>

<keyword, "int">, <int,"2">, <id,"x">, <symbol," ;">

**String str =1;**

نوع خطا عبارت است از:

۱ یک عدد صحیح (int) است.

باید یک رشته (String) باشد و عدد نباشد.

و عدد صحیح مستقیماً قابل تبدیل به String نیست مگر با تبدیل صریح (مانند "1")



# مثال

## Lexeme

لکسم یک واحد واقعی و مشهود در کد منبع است. در عبارت  $sum = 3 + 2$ ;  $sum$  نیز قسمتهایی مثل؛  $3 + 2$  هر کدام یک لکسم هستند.

## Token

برای اعداد NUMBER / برای عملگر انتساب ASSIGN\_OP / برای شناسه‌ها IDENTIFIER داریم.

Int X;

استفاده از کاراکتر فارسی (مانند ج) در جایی که معمولاً انتظار می‌رود کلمات کلیدی انگلیسی باشند (مانند int)

ترکیب X؛ بدون تعریف یا مقداردهی مشخص.

ج Int یک کلمه کلیدی معتبر در هیچ زبان برنامه‌نویسی استانداردی نیست.

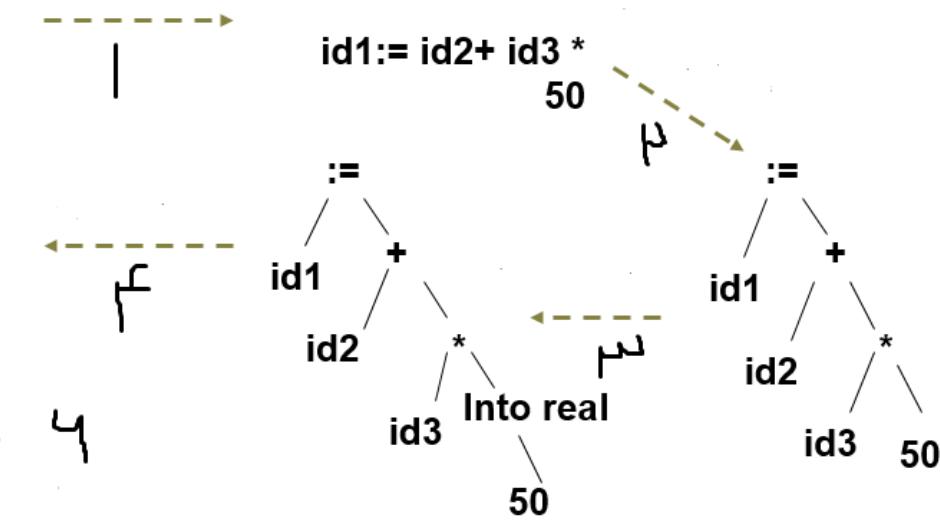


# فرآیند تبدیل یک عبارت سطح بالا به کد نهایی

Area:= Pos + Rate \* 50

```
tem1:=into real 50
tem2:=id3 * tem1
tem3:= id2 + tem2
Id1:= tem3
```

```
tem1:= id3 * 50.0
Id1:= id2 + tem1
```



```
Mov id3, R2           ; Rate → R2
Mul 50.0, R2          ; R2 = R2 * 50.0
Mov id2, R1           ; Pos → R1
Add R2, R1            ; R1 = R1 + R2
Mov R1, id1           ; نتیجه → Area
```

Stage Number	Stage Name	Description
1	Lexical Analyzer	Tokenizes the source code
2	Syntax Analyzer	Parses tokens according to grammar
3	Semantic Analyzer	Checks semantic correctness
4	Intermediate Code Generator	Produces intermediate representation
5	Optimizer	Improves intermediate code
6	Code Generator	Generates final machine code

# طبقه بندی کامپایلرهای

## جدول ۱. انواع کامپایلر

نام	توضیح
One-pass	کامپایلر فقط یک بار کد منبع را می‌خواند و تبدیل می‌کند. این روش سریع‌تر است اما محدودیت‌هایی دارد؛ نمی‌تواند به راحتی به اطلاعاتی که بعداً می‌آید دسترسی داشته باشد. مناسب برنامه‌های ساده‌تر یا زبان‌هایی با ساختار خطی.
Multi-pass	کامپایلر چند بار کد منبع را می‌خواند و پردازش می‌کند. هر مرحله وظیفه خاصی دارد (مثلاً تحلیل لغوی، تحلیل نحوی، تحلیل معنایی، بهینه‌سازی و تولید کد). این روش انعطاف‌پذیرتر و قدرتمندتر است و معمولاً برای زبان‌های پیچیده‌تر استفاده می‌شود.
Load-&Debugger and-go	در این روش برنامه پس از کامپایل به صورت فوری اجرا می‌شود. معمولاً برای محیط‌های توسعه تعاملی (IDE) یا زبان‌های تفسیر شده استفاده می‌شود. مزیت: دیباگ سریع و آسان؛ معایب: سرعت اجرای کمتر نسبت به کد کامپایل شده کامل.
Optimizer	بخشی از کامپایلر یا ابزار جانبی که کد تولید شده را تحلیل و اصلاح می‌کند تا سریع‌تر یا کم حجم‌تر اجرا شود. مثلاً حذف کدهای تکراری، ساده‌سازی عملیات‌ها، کاهش مصرف حافظه و ... که باعث بهبود کارایی نهایی برنامه می‌شود.

## تمرین ۲

تبدیل عبارت سطح بالا به کد نهایی با ریجسترها فرضی را انجام دهید.

مهلت تحویل: ۱۱ مهرماه ۱۴۰۴

موفق باشید



دانشگاه شهید بهشتی کرمان

# به نام دانای نیآموخته

درس: کامپیویلر

ترم: نیمسال اول (۰۴-۰۵)

دپارتمان: مهندسی کامپیویتیر

مدرس: دکتر شیما شفیعی

# گرامر

- زبان: مجموعه ای از رشته ها
- رشته: یک دنباله از الفبا
- الفبا: یک مجموعه از سymbol ها

$\Sigma\{a, b\} \rightarrow String: abab, bba, aa, \dots$

# گرامر

تعریف: مجموعه قوانینی که به کمک آن ها می توان رشته های یک زبان را ساخت.

$$G = \langle V, T, S, P \rangle$$

**V= Variables (NonTerminals)**

**T= Terminals**

**S= Start Symbol**

**P= Set of Production Rules**

## مثال

$$V = \{ G, H \}$$

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

$$S = G$$

$$P = \{ G \rightarrow H * \text{id} , H \rightarrow H / G , H \rightarrow H + H \}$$

String: id \* id+ id / id

## مثال

$\text{<Sentence>} \rightarrow \text{<Subject>} \text{ <Verb>} \text{ <Object>}$

Variable

$\text{<Subject>} \rightarrow \text{<Pronoun>} \mid \text{<Noun>}$

Terminal

$\text{<Pronoun>} \rightarrow \text{i} \mid \text{we} \mid \text{you} \mid \text{he} \mid \text{she} \mid \text{it} \mid \text{they}$

$\text{<Verb>} \rightarrow \text{open} \mid \text{close}$

$\text{<Object>} \rightarrow \text{door} \mid \text{window}$

ادامه

**<Text> ⇒ <Sentence>**

⇒ **<Subject> <Verb> <Object>**

⇒ **<Pronoun> <Verb> <Object>**

⇒ **i <Verb> <Object>**

⇒ **i close <Object>**

⇒ **i close window**

**String = i close window**

## مثال

$V = \{S, A, B\}$

$P = S \rightarrow AB$

$T = \{a, b\}$

$A \rightarrow aAb \mid \lambda$

$B \rightarrow Bb \mid b$

$a^2 b^3 = ?$

$S \rightarrow AB \Rightarrow aAbB \Rightarrow aaAbbB \Rightarrow aabbB \Rightarrow^+ a^2 b^3$

Derivational

$\lambda: lambda$

$|\lambda| = 0$

## مثال

$\Rightarrow^*$  اشتقاق (Derivation) با صفر گام یا بیشتر

$\Rightarrow^+$  اشتقاق با یک گام یا بیشتر

$$S \Rightarrow^* aAbB$$

$$S \Rightarrow^+ aAbB$$

$$S \Rightarrow^* aabb$$

$$S \Rightarrow^* S$$

$$S \Rightarrow^+ S \quad \times$$

•

## گرامر

زبان گرامر: مجموعه رشته هایی که آن گرامر ایجاد می کند

$$L(G) = \{w \in T^* \mid S \Rightarrow^* w\}$$

$$V = \{S, A\}$$

$$T = \{a, b\}$$

$$P = \{ S \rightarrow A, A \rightarrow aAb \mid \lambda \}$$



$$L(G) = \{a^n b^n \mid n \geq 0\}$$

زبان گرامر G برابر است با مجموعه‌ای از رشته‌ها w که از حروف نهایی (ترمینال‌ها) ساخته شده‌اند، به طوری که نماد آغازین S بتواند در صفر یا چند گام به w مشتق شود.

## انواع گرامر چامسکی

۱- گرامرهای نوع سوم (منظم Regular)

۲- گرامرهای نوع دوم (مستقل از متن Context Free)

۳- گرامرهای نوع اول (حساس به متن Context Sensitive)

۴- گرامرهای نوع صفر (بی قید و شرط Unrestricted)

# گرامرهای منظم (Regular)

منظم از راست

از راست: در هر قدم انجام جایگزینی روی سمت راست ترین غیرپایانه

$$\begin{cases} A \rightarrow X \\ A \rightarrow XB \end{cases}$$

$$x \in T^*, B \in V$$

منظم از چپ

از چپ: در هر قدم انجام جایگزینی روی سمت چپ ترین غیرپایانه

$$\begin{cases} A \rightarrow X \\ A \rightarrow BX \end{cases}$$

$$x \in T^*, B \in V$$

## گرامر نامنظم

Example of Non-Regular Grammar	Generated Language	Reason for Non-Regularity
$S \rightarrow aSb \mid ab$	$\{a^n b^n \mid n \geq 1\}$	Nonterminal $S$ appears in the middle of the string (not regular).
$S \rightarrow SS \mid a$	$\{a^n \mid n \geq 1\}^*$ (concatenation language)	Rule $S \rightarrow SS$ has consecutive nonterminals; not regular.

## گرامرهای مستقل از متن (Context Free)

$$A \rightarrow \alpha$$

$$\alpha \rightarrow (V \cup T)^*$$

$$A \in V$$

مثال:

$$L = \{a^n b^n \mid n \geq 0\}$$

$$S \rightarrow aSb \mid \lambda$$

مثال:

$$L = \{w \mid n_a(w) = n_b(w)\}$$

$$S \rightarrow aSb \mid bSa \mid \lambda$$

$$S \rightarrow SS$$

## گرامرهاي مستقل از متن (Context Free)

زبان مستقل از متن (Context-Free Language) به زبان‌هایی گفته می‌شود که توسط یک گرامر مستقل از متن (Grammer – CFG) تولید می‌شوند. در این نوع گرامرها، هر قاعده‌ی تولید به صورت  $A \rightarrow -\lambda$  است که در آن  $A$  یک غیرترمینال و  $\lambda$  ترکیبی دلخواه از نمادهای ترمینال و غیرترمینال است. زبان‌های مستقل از متن توانایی بیان ساختارهای تودرتو و بازگشتی مانند پرانتزهای جفت‌شده، عبارات ریاضی، یا ساختارهای نحوی زبان برنامه نویسی را دارند.

زبان محدود (Finite Language) زبانی است که شامل تعداد محدودی از رشته‌ها است؛ یعنی مجموعه‌ی رشته‌های آن پایان پذیر است.

**Finite Languages ⊂ Regular Languages ⊂ Context-Free Languages**

Finite Language

$$L = \{ \text{str}(n) \mid 101 \leq n \leq 999 \}$$

Regular Language

$$L = \{a^* b\} = \{b, ab, aab, aaab, \dots\}$$

Context-Free Language

$$L = \{a^n b^n \mid n \geq 0\} = \{b, ab, aab, aaab, \dots\}$$

## تمرین ۳

$$L = \{a^n b^m \mid n \geq 1, m \geq 1\}$$

تشخیص نوع زبان

توضیح دلیل انتخاب نوع

نوشتن گرامر مستقل از متن

مهلت تحویل: ۱۱ مهرماه ۱۴۰۴

موفق باشد



# به نام دانای نیآموخته

درس: کامپایلر

ترم: نیمسال اول (۰۴-۰۵)

دپارتمان: مهندسی کامپیوتر

مدرس: دکتر شیما شفیعی

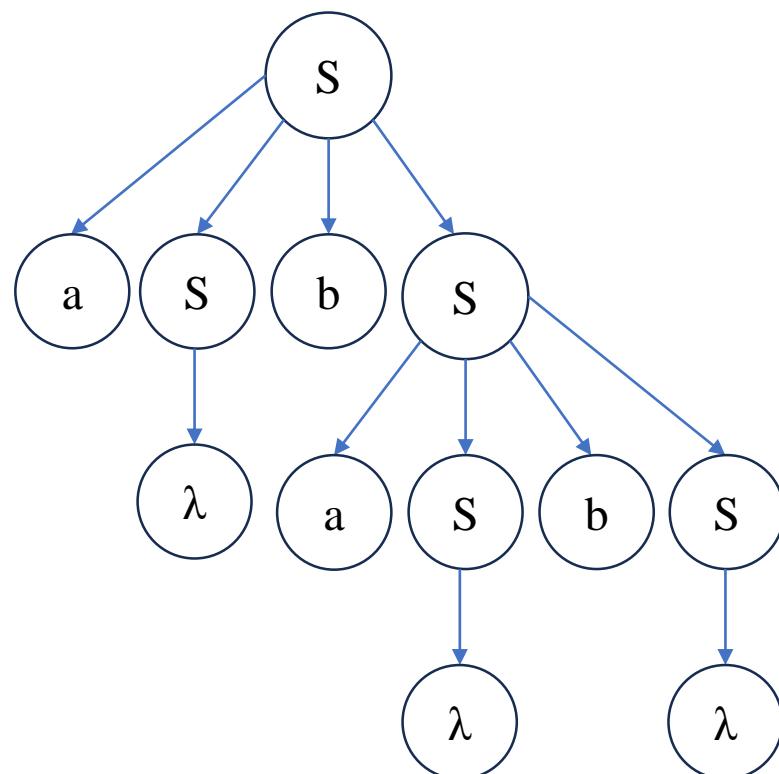
## اشتقاق

$S \Rightarrow aSbS \Rightarrow aSbaSbS \Rightarrow aSbaSb \Rightarrow aSbab \Rightarrow abab$

$S \Rightarrow aSbS \Rightarrow abS \Rightarrow abaSbS \Rightarrow ababS \Rightarrow abab$

اشتقاق به راست

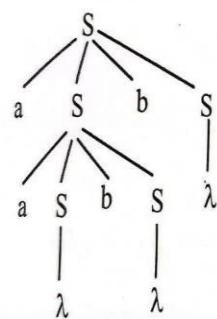
اشتقاق به چپ



Derivation tree

## گرامر مبهم

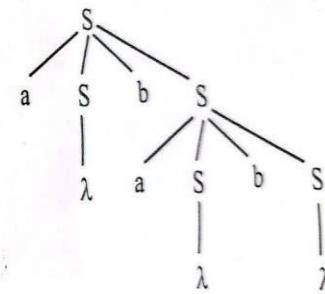
$1 = G \& 1 < Tree \rightarrow \text{Lexical Analyzer}$



Ambiguous

$$G: S \rightarrow aSbS \mid bSaS \mid \lambda$$

String : abab



Ambiguous

$$G: S \rightarrow aS \mid a \mid \lambda$$

String : a\*

$$S \rightarrow aS \mid \lambda$$



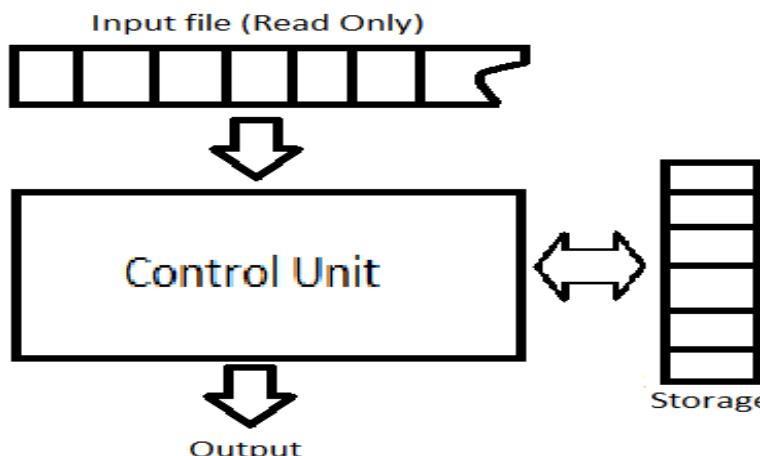
# خودکاره (Automata)

## انواع آتماتا:

۱- پذیرنده (Acceptor): آتماتایی که خروجی آن قبول یا عدم قبول است (ورودی خود را قبول یا رد می کند).

۲- مبدل (Transducer): آتماتایی که خروجی آن به صورت رشته است.

- داده‌ها ابتدا از یک فایل ورودی خوانده می‌شوند که حالت فقط خواندنی دارد.
- این داده‌ها به واحد کنترل ارسال می‌شوند.
- واحد کنترل مسئول پردازش یا کنترل داده‌هاست.
- داده‌های پردازش شده یا نتایج توسط واحد کنترل به عنوان خروجی ارائه می‌شوند.
- همچنین، واحد کنترل به حافظه متصل است و می‌تواند داده‌ها را از حافظه بخواند یا به آن بنویسد.



شکل ۱. شماتیک یک آتماتا

## ادامه

### 1- Turing Machine

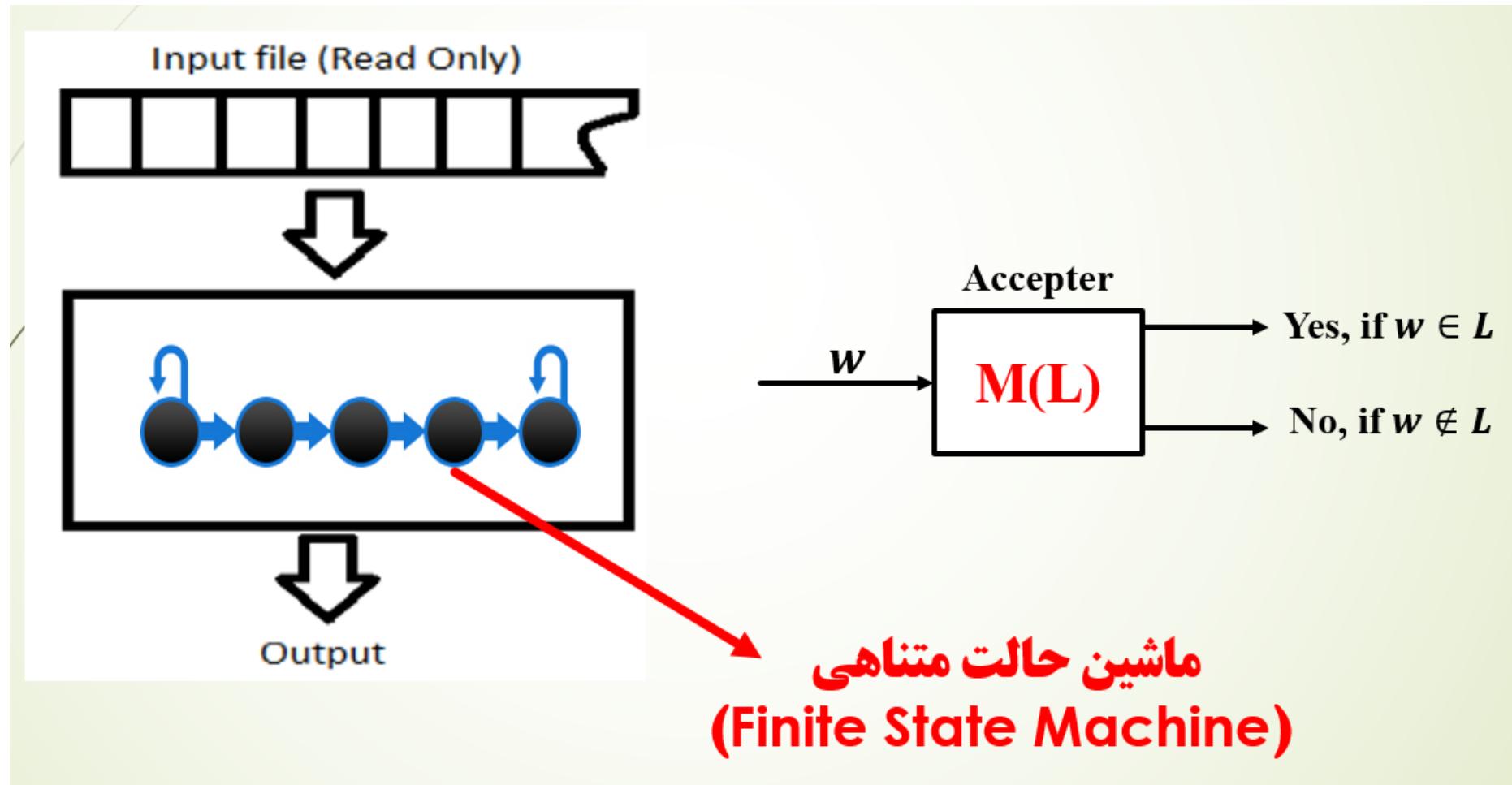
### 2- Linear bounded automata (LBA)

### 3- Push-Down automata (PDA)

### 4- Finite State automata (FSA)

اتوماتا	حافظه دارد؟	نوع حافظه
Turing Machine	بله	نوار (بینهایت)
Linear Bounded Automata	بله	نوار محدود (خطی)
Push-Down Automata (PDA)	بله	پشته
Finite State Automata (FSA)	خیر (بی حافظه)	(State)

# ماشین متناهی



EX: Lexical Analyzer

## ماشین متناهی

Regular Language

$$M = (Q, \Sigma, \delta, q_0, F)$$

$Q$  = مجموعه حالات ( $q_0, q_f$ )

$\Sigma$  = الفبای زبان (غیر پایانه)

$\delta$  = قابع گذار (مجموعه دستور العمل برای پذیرش یا ریجکت رشته)

$q_0$  = حالت شروع

$F \subseteq Q$  = مجموعه حالات نهایی ( $F$ )

## ماشین متناهی

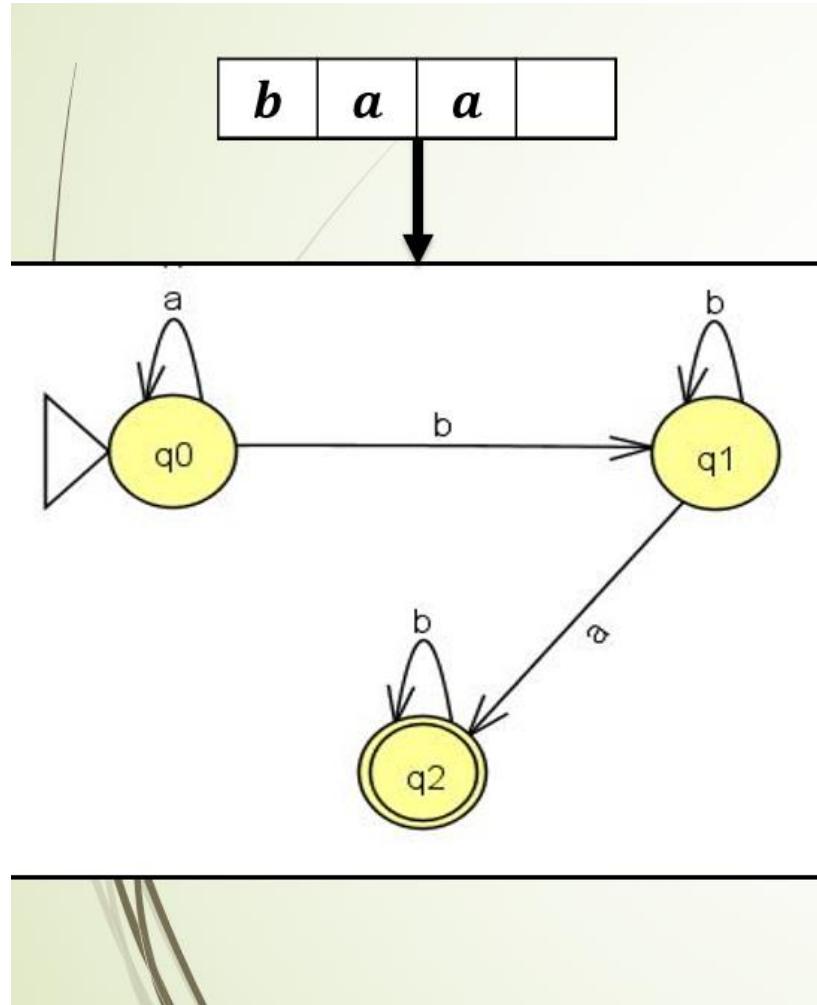
۱- ماشین متناهی قطعی (Deterministic FA) DFA

۲- ماشین متناهی غیر قطعی (Non-Deterministic FA) NFA

در DFA، برای هر حالت و هر ورودی فقط یک مسیر یکتا وجود دارد و رفتار ماشین کاملاً مشخص است.

در NFA، برای هر حالت و ورودی می‌تواند چندین مسیر مختلف وجود داشته باشد یا حتی هیچ مسیری نباشد؛ همچنین ممکن است انتقال بدون ورودی ( $\epsilon$ -transition) نیز باشد.

## ادامه



$$Q = \{q_0, q_1, q_2\}$$

$$\text{start}(q_0) = q_0$$

$$\begin{aligned}\delta = (q_0, a) &= q_0 \\ \delta = (q_0, b) &= q_1 \\ \delta = (q_1, a) &= q_2 \\ \delta = (q_1, b) &= q_1 \\ \delta = (q_2, b) &= q_2\end{aligned}$$

$$\delta: Q \times \Sigma \rightarrow Q$$

$$\Sigma = \{a, b\}$$

$$F = \{q_2\}$$

Transition Table

$\delta$	$a$	$b$
$q_0$	$q_0$	$q_1$
$q_1$	$q_2$	$q_1$
$q_2$	$\times$	$q_2$

## ادامه

$$L(M) = \{w | w \in \Sigma^*, \delta^*(q_0, w) \in F\}$$

بنابراین این زبان شامل تمام رشته‌هایی است که اگر آنها را به DFA بدهیم، از  $q_0$  شروع کرده و پس از پردازش رشته  $w$ ، در یکی از حالت‌های نهایی  $F$  قرار می‌گیرد.

$$\overline{L(M)} = \{w | w \in \Sigma^*, \delta^*(q_0, w) \notin F\}$$

شامل تمام رشته‌هایی است که DFA آنها را قبول نمی‌کند.

یعنی پس از پردازش رشته، ماشین در حالت نهایی قرار نمی‌گیرد.

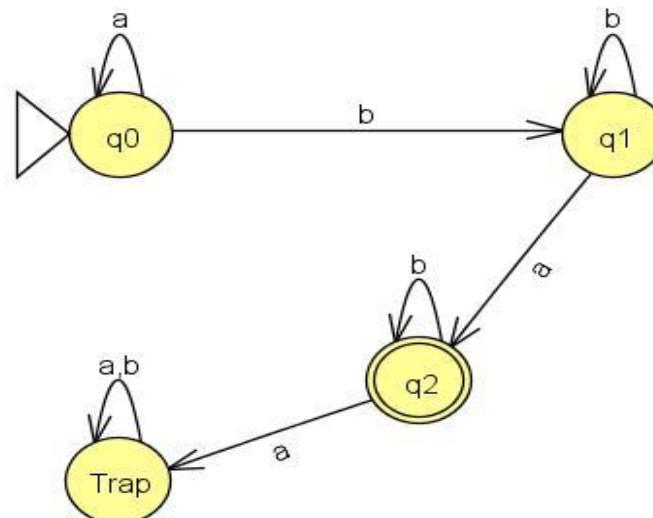
$w = bbb$

# Trap

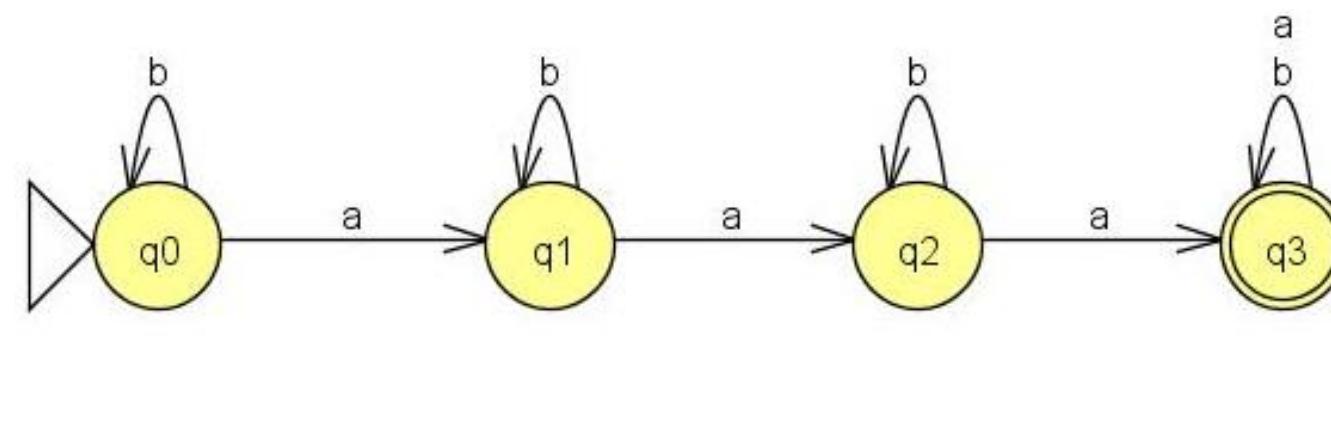
تعريف حالت Trap: یک حالت غیر نهایی است که دنباله‌ی رشته را مصرف می‌کند تا رشته

به اتمام برسد

$$Q = \{q_0, q_1, q_2, \text{Trap}\}$$



# مثال

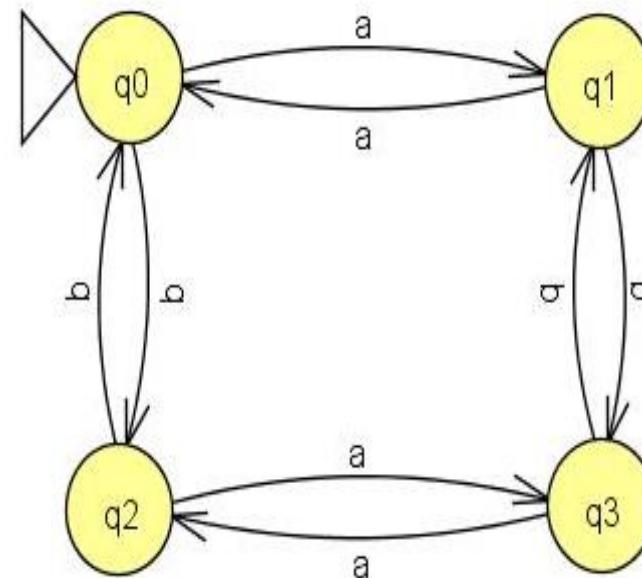


$$L = \{w \mid n_a(w) \geq 3\}$$

$$L(M_1) = \emptyset$$

اما چون این ماشین هیچ حالت نهایی نداره

هیچ رشته‌ای، حتی رشته‌ی تهی ( $\epsilon$ )، هم پذیرفته نمی‌شه.

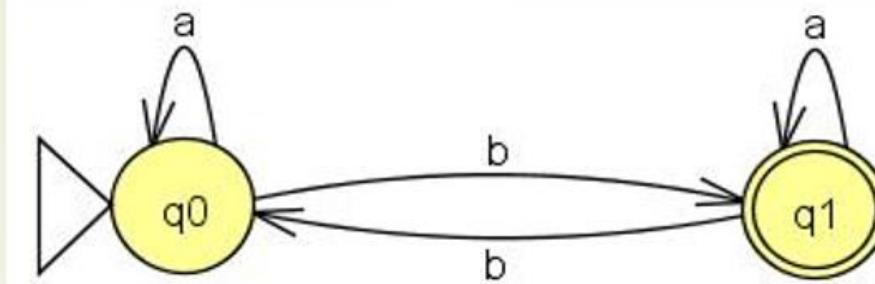


## مثال

**تعريف DFA:** به یک DFA یک FSA گفته می شود اگر در هر کدام از حالت های آن به ازای هر کدام از الفبا دقیقا یک حرکت وجود داشته باشد.

**تعريف NFA:** به یک NFA یک FSA گفته می شود اگر حداقل در یکی از حالت های آن به ازای حداقل یکی از الفبا بیش از یک حرکت وجود داشته باشد یا حرکت وجود نداشته باشد.

## مثال

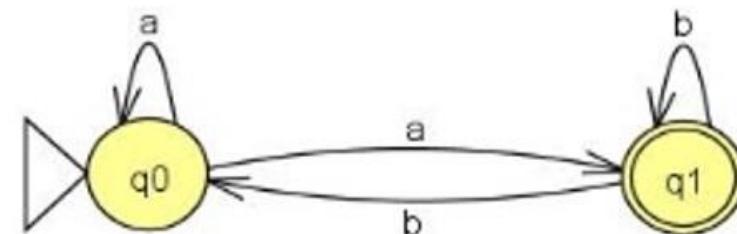


$$\delta = (q_0, a) = q_0$$

$$\delta = (q_0, b) = q_1$$

$$\delta = (q_1, a) = q_1$$

$$\delta = (q_1, b) = q_0$$



$$\delta = (q_0, a) = \{q_0, q_1\}$$

$$\delta = (q_0, b) = \{\}$$

$$\delta = (q_1, a) = \{\}$$

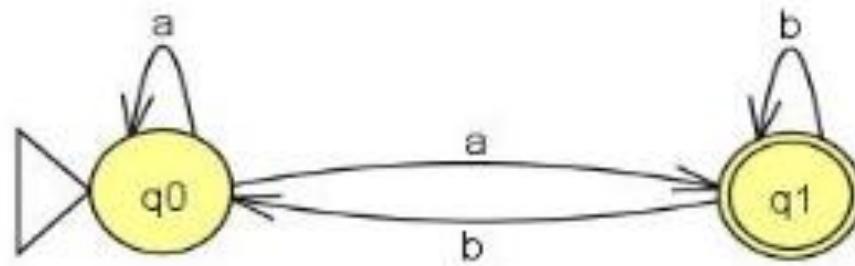
$$\delta = (q_1, b) = \{q_0, q_1\}$$

Transition Function

$$\delta : Q \times \Sigma \rightarrow Q$$

$$\delta : Q \times \Sigma \rightarrow 2^Q$$

## مثال



$$\delta^* = (q_0, abb) = \{q_0, q_1\}$$

$$\delta^* = (q_0, ba) = \{\}$$

این رشتہ تعریف نشده و در  $q_0$  متوقف می شود

## آutomاتا متناهی غیر قطعی (NFA)

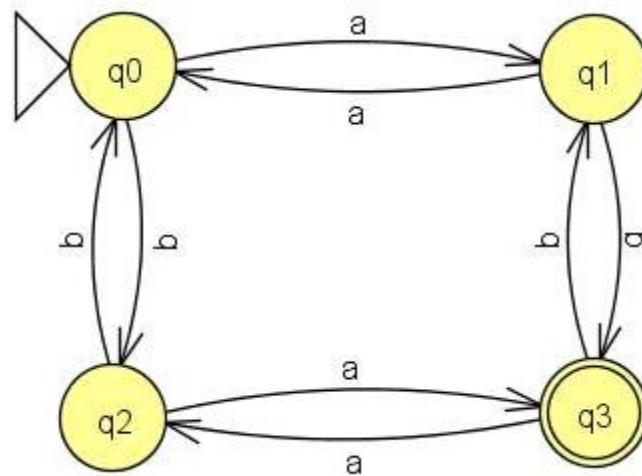
تعریف پذیرش در **NFA**: اگر حداقل یک مسیر وجود داشته باشد که بتوان با پیمایش آن پس از اتمام رشته در حالت نهایی متوقف شد آن رشته پذیرفته می شود (نوارخوان به انتهای رشته رسیده باشد) و عدم پذیرش یک رشته زمانی است که هیچ مسیری به صورت بالا وجود نداشته باشد.

تعریف رسمی پذیرش **NFA**:

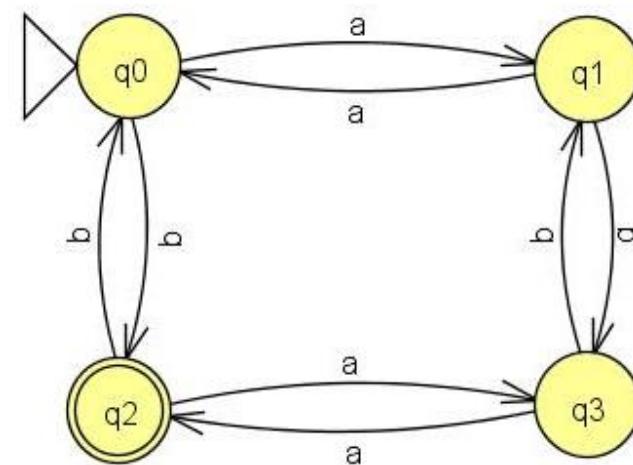
$$L(M) = \{w | w \in \Sigma^*, \delta^*(q_0, w) \cap F \neq \emptyset\}$$

$$\overline{L(M)} = \{w | w \in \Sigma^*, \delta^*(q_0, w) \cap F = \emptyset\}$$

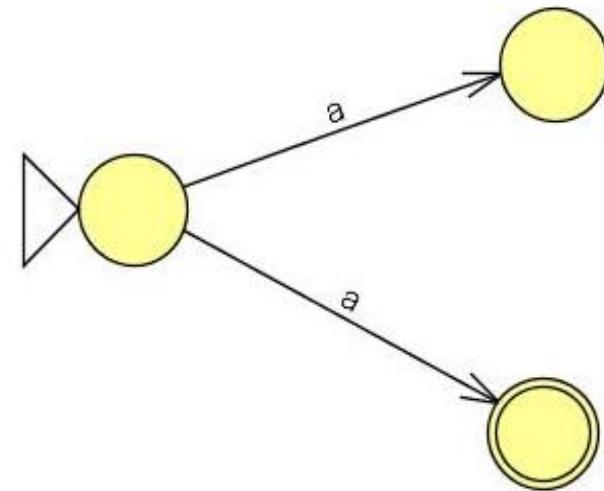
## تمرین ۴



a) Type , L (M) ?



b) Type , L (M) ?



c) Type , L (M) ?

مهلت تحویل: ۱۴۰۴ مهرماه

موفق باشید



# به نام دانای نیآموخته

درس: کامپایلر

ترم: نیمسال اول (۰۴-۰۵)

دپارتمان: مهندسی کامپیوتر

مدرس: دکتر شیما شفیعی

# Programming Language Specs

- Since the 1960s, the syntax of every significant programming language has been specified by a formal grammar
  - First done in 1959 with BNF (Backus-Naur Form), used to specify ALGOL 60 syntax
  - Borrowed from the linguistics community (Chomsky)

## Positive integer

```

<positive-integer> ::= <non-zero-digit> <digits>
<digits>      ::= ε | <digit> <digits>
<digit>       ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
<non-zero-digit> ::= "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"

```

$(2+3)* 4+5$

```

<expr>   ::= <term> | <term> + <expr>
<term>   ::= <factor> | <factor> * <term>
<factor> ::= ( <expr> ) | <number>
<number> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

```

`number` → 2, 3, 4, 5

`factor` → (2 + 3), 4, 5

`term` → (2 + 3) \* 4

`expr` → (2 + 3) \* 4 + 5

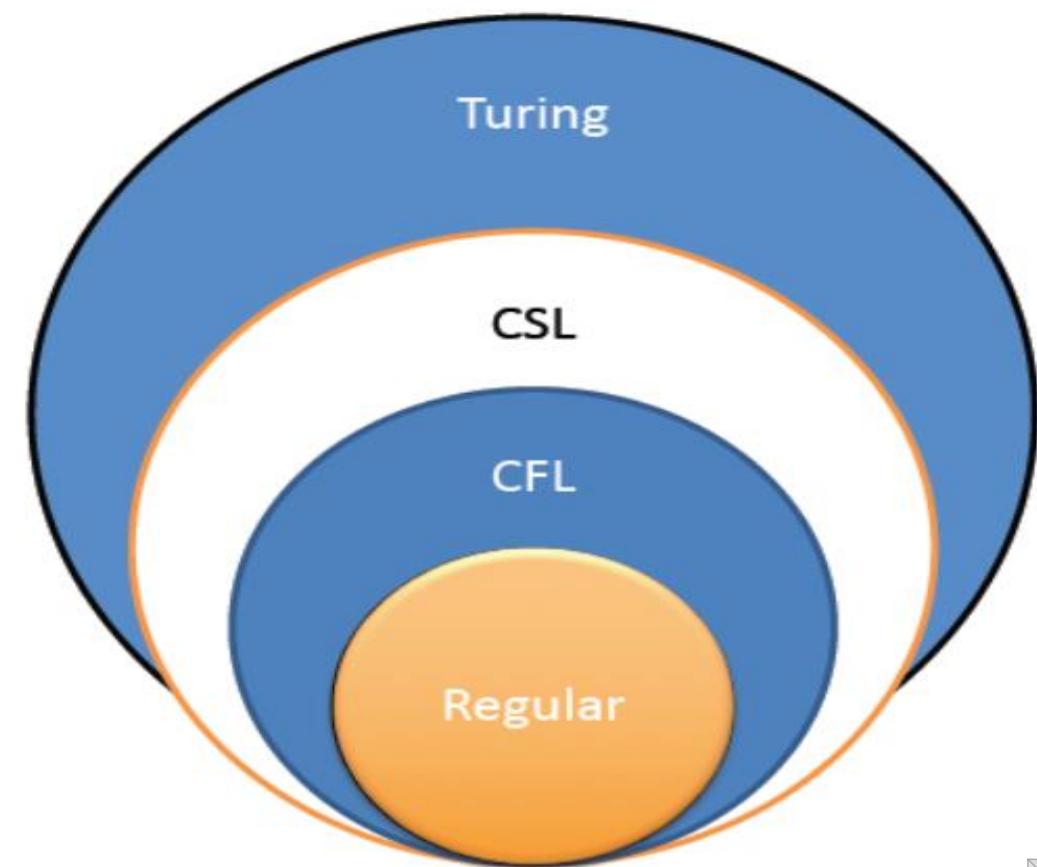
- عبارات ریاضی‌ای را تولید می‌کند که شامل + هستند.
- بخش‌هایی از عبارت هستند که شامل \* هستند.
- می‌تواند یک عدد یا یک عبارت داخل پرانتز باشد.
- ارقام ۰ تا ۹ هستند.

# Formal Languages & Automata Theory (a review on one slide)

- Alphabet: a finite set of symbols and characters
- String: a finite, possibly empty sequence of symbols from an alphabet
- Language: a set of strings (possibly empty or infinite)
- Finite specifications of (possibly infinite) languages
  - Automaton – a recognizer; a machine that accepts all strings in a language (and rejects all other strings)
  - Grammar – a generator; a system for producing all strings in the language (and no other strings)
- A particular language may be specified by many different grammars and automata
- A grammar or automaton specifies only one language

## Language (Chomsky) hierarchy: quick reminder

- Regular (Type-3) languages are specified by regular expressions/grammars and finite automata (FSAs)
  - Specs and implementation of scanners
- Context-free (Type-2) languages are specified by context-free grammars and pushdown automata (PDAs)
  - Specs and implementation of parsers
- Context-sensitive (Type-1) languages ... aren't too important (at least for us)
- Recursively-enumerable (Type-0) languages are specified by general grammars and Turing machines



*program ::= statement | program statement*

*statement ::= assignStmt | ifStmt*

*assignStmt ::= id = expr ;*

*ifStmt ::= if ( expr ) statement*

*expr ::= id | int | expr + expr*

*id ::= a | b | c | i | j | k | n | x | y | z*

*int ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9*

Grammar Rule	Example
<b>program ::= statement</b>	a = 5 ;
<b>program ::= program statement</b>	a = 5 ; if ( a ) b = 3 ;
<b>statement ::= assignStmt</b>	x = 4 ;
<b>statement ::= ifStmt</b>	if ( 1 ) y = 2 ;
<b>assignStmt ::= id = expr ;</b>	z = a + 1 ;
<b>ifStmt ::= if ( expr ) statement</b>	if ( b ) k = 2 ;
<b>expr ::= id</b>	c
<b>expr ::= int</b>	7
<b>expr ::= expr + expr</b>	i + 3
<b>id ::= a</b>	a
<b>int ::= 1</b>	1

# Exercise: Derive a simple program

```
program ::= statement | program statement  
statement ::= assignStmt | ifStmt  
assignStmt ::= id = expr ;  
ifStmt ::= if ( expr ) statement  
expr ::= id | int | expr + expr  
id ::= a | b | c | i | j | k | n | x | y | z  
int ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

a = 1 ;      if ( a + 1 )      b = 2 ;

# Productions

- The rules of a grammar are called productions
- Rules contain:
  - Nonterminal symbols: grammar variables (program, statement, id, etc.)
  - Terminal symbols: concrete syntax that appears in programs (a, b, c, 0, 1, if, =, (, ), ...)
- Meaning of:  
 $\text{nonterminal} ::= \langle \text{sequence of terminals and nonterminals} \rangle$   
In a derivation, an instance of nonterminal can be replaced by the sequence of terminals and nonterminals on the right of the production
- Often there are several productions for a nonterminal – can choose any in different parts of derivation

## گرامر برای برنامه

```
a = 1 ;  
  
program      ::= statement  
statement    ::= assignStmt  
assignStmt   ::= id = expr ;  
expr         ::= int  
id           ::= a  
int          ::= 1
```

## گرامر برای برنامه

```
a = 1 ; if ( a + 1 ) b = 0 ;  
  
program      ::= statement | program statement  
statement    ::= assignstmt | ifstmt  
assignstmt   ::= id = expr ;  
ifstmt       ::= if ( expr ) statement  
expr         ::= id | int | expr + expr  
id           ::= a | b  
int          ::= 0 | 1
```

## تمرین ۵

- if  $(x+1)y=0;$
- گرامر برای دو برنامه آورده شده نیز بنویسید.
- if  $(y+0);$  if  $(x)x=1;$

مهلت تحویل: ۲۵ مهرماه ۱۴۰۴

موفق باشید



# به نام دانای نیآموخته

درس: کامپایلر

ترم: نیمسال اول (۰۴-۰۵)

دپارتمان: مهندسی کامپیوتر

مدرس: دکتر شیما شفیعی

# Derivation

Using a language grammar (such as BNF) to achieve a program.

a = 1 ; if ( a + 1 ) b = 2 ;

```
<program>
→ <program> <statement>
→ <statement> <statement>

<statement> → <assignStmt>
→ a = <expr> ;
→ a = 1 ;

<statement> → <ifstmt>
→ if ( <expr> ) <statement>
→ if ( a + 1 ) <statement>
→ if ( a + 1 ) b = 2 ;
```

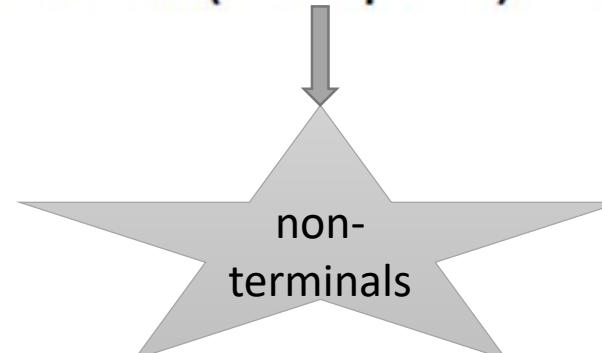
# Alternative Notations

- There are several notations for productions in common use; all mean the same thing

*ifStmt ::= if ( expr ) statement*

*ifStmt → if ( expr ) statement*

*<ifStmt> ::= if ( <expr> ) <statement>*



Notation Style	Description
<code>::=</code>	Common in programming language documentation and grammar definitions
<code>→</code>	Used in formal language theory and derivation steps
<code>&lt;&gt;</code> (BNF form)	Official Backus-Naur Form; non-terminals are enclosed in angle brackets

```
if (a + 1) b = 2;
```

```
ifstmt ::= if ( expr ) statement  
expr ::= id + int  
statement ::= id = int ;
```

```
ifstmt → if ( expr ) statement  
expr → id + int  
statement → id = int ;
```

```
<ifstmt> ::= if ( <expr> ) <statement>  
<expr> ::= <id> + <int>  
<statement> ::= <id> = <int> ;  
<id> ::= a | b  
<int> ::= 1 | 2
```

- Parsing

Parsing: reconstruct the derivation (syntactic structure) of a program.

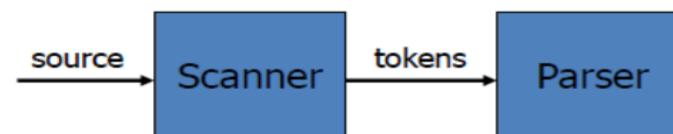
In principle, a single recognizer could work directly from a concrete, character-by-character grammar.

In practice, this is never done.

In theory, it is possible to parse directly from character . Still,in practice this method is very inefficient, so we always do the parsing after the lexing step, using tokens that have been prepared in advance.

# Parsing & Scanning

- In real compilers the recognizer is split into two phases
  - Scanner: translate input characters to tokens
    - Also, report lexical errors like illegal characters and illegal symbols
  - Parser: read token stream and reconstruct the derivation



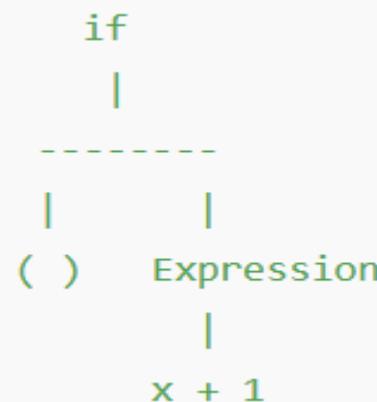
Main Responsibility	Output	Input	Other Name	Phase
Identify meaningful units and report lexical errors	Tokens	Characters	Lexical Analysis	Scanning
Check grammatical structure and build syntax tree	Parse Tree / AST	Tokens	Syntax Analysis	Parsing

if (x + 1)

## tokens

IF LPAREN IDENTIFIER(x) PLUS NUMBER(1) RPAREN

## syntax tree or AST



# Why Separate the Scanner and Parser?

- Simplicity & Separation of Concerns
  - Scanner hides details from parser (comments, whitespace, input files, etc.)
  - Parser is easier to build; has simpler input stream (tokens) and simpler interface for input
- Efficiency
  - Scanner recognizes regular expressions – proper subset of context free grammars
    - (But still often consumes a surprising amount of the compiler's total execution time)

# But ...

- Not always possible to separate cleanly
- Example: C/C++/Java *type* vs *identifier*
  - Parser would like to know which names are types and which are identifiers, but...
  - Scanner doesn't know how things are declared
- So we hack around it somehow...
  - Either use simpler grammar and disambiguate later, or communicate between scanner & parser
  - Engineering issue: try to keep interfaces as simple & clean as possible

Scanner can't tell if a name is a type or identifier

Because it lacks access to declarations or symbol tables

Parser needs that info to disambiguate syntax

Especially in languages like C/C++/Java

Workarounds

Use simpler grammar and resolve later, or allow limited scanner-parser communication

class Foo {};

Foo x;



type

# Typical Tokens in Programming Languages

- Operators & Punctuation
  - + - \* / ( ) { } [ ] ; : :: < <= == != ! ...
  - Each of these is a distinct lexical class
- Keywords
  - if while for goto return switch void ...
  - Each of these is also a distinct lexical class (*not a string*)
- Identifiers
  - A single ID lexical class, but parameterized by actual id
- Integer constants
  - A single INT lexical class, but parameterized by int value
- Other constants, etc.

Token Type	Example	Lexical Class	Description
Operators & Punctuation	+ , == , { , ;	PLUS , EQ , LBRACE , SEMICOLON	Each symbol is its own lexical class
Keywords	if , while , return	IF , WHILE , RETURN	Reserved words with specific meaning in grammar
Identifiers	x , count , sum	ID("x") , ID("count")	One lexical class ( ID ), parameterized by name
Integer Constants	0 , 123 , -5	INT(0) , INT(123)	One lexical class ( INT ), parameterized by value
Other Constants	"hi" , 'a' , 3.14	STRING("hi") , CHAR('a') , FLOAT(3.14)	Various constant types with attached values

# Principle of Longest Match

- In most languages, the scanner should pick the longest possible string to make up the next token if there is a choice
- Example

return maybe != iffy;

should be recognized as 5 tokens

RETURN

ID(maybe)

NEQ

ID(iffy)

SCOLON

i.e., != is one token, not two; “iffy” is an ID, not IF followed by ID(fy)

## تمرین ۶

Program → StmtList

StmtList → StmtList Stmt | ε

Stmt → "if" Expr "then" Stmt "else" Stmt | "while" Expr "do" Stmt | "{" StmtList  
"}" | ID "=" Expr ";"

Expr → Expr "+" Term | Term

Term → Term "\*" Factor | Factor

Factor → "(" Expr ")" | ID | NUM

الف) لیستی از توکن‌ها را بنویسید

ب) یک AST برای آن ترسیم کنید

پ) عبارتی طراحی کنید که اگر اسکنر قانون "برداشت بلندترین رشته ممکن" را رعایت نکند، غلط تفسیر شود یا ابهام ایجاد کند.

مهلت تحويل: ۲۵ مهرماه ۱۴۰۴

موفق باشید



# به نام دانای نیآموخته

درس: کامپایلر

ترم: نیمسال اول (۰۵-۰۴)

دپارتمان: مهندسی کامپیووتر

مدرس: دکتر شیما شفیعی

# Lexical Complications

- Most modern languages are free-form
  - Layout doesn't matter
  - Whitespace separates tokens
- Alternatives
  - Fortran – line oriented
  - Haskell, Python – indentation and layout can imply grouping
- And other confusions
  - In C++ or Java, is `>>` a shift operator or the end of two nested templates or generic classes?

layout

```
int x=5;  
int y = 10;
```

```
int x = 5; int y = 10;
```

Python

```
for i in range(5):  
    print("Number:", i)  
print("Loop finished")
```

C++: shift operator

```
int x = a >> 2;
```

```
std::vector<std::vector<int>> matrix;
```

# Regular Expressions and FAs

- The lexical grammar (structure) of most programming languages can be specified with regular expressions  
(Sometimes a little cheating is needed)
- Tokens can be recognized by a deterministic finite automaton
  - Can be either table-driven or built by hand based on lexical grammar

# Regular Expressions

- Defined over some alphabet  $\Sigma$ 
  - For programming languages, alphabet is usually ASCII or Unicode
- If  $re$  is a regular expression,  $L(re)$  is the language (set of strings) generated by  $re$

# Fundamental REs

$re$	$L(re)$	Notes
a	{ a }	Singleton set, for each a in $\Sigma$
$\epsilon$	{ $\epsilon$ }	Empty string
$\emptyset$	{ }	Empty language

# Operations on REs

$re$	$L(re)$	Notes
$r s$	$L(r)L(s)$	Concatenation
$r   s$	$L(r) \cup L(s)$	Combination (union)
$r^*$	$L(r)^*$	0 or more occurrences (Kleene closure)

- Precedence: \* (highest), concatenation, | (lowest)
- Parentheses can be used to group REs as needed
- In “real” regular expression computer tools, need some way to “escape” literal ‘\*’ or ‘|’ characters vs. operators – but don’t worry, or use different fonts, for math regexps.

# Examples

<i>re</i>	Meaning
+	single + character
!	single ! character
=	single = character
!=	2 character sequence "!="
xyzzy	5 character sequence "xyzzy"
(1 0)*	0 or more binary digits
(1 0)(1 0)*	1 or more binary digits
0 1(0 1)*	sequence of binary digits with no leading 0's, except for 0 itself

# Abbreviations

- The basic operations generate all possible regular expressions, but there are common abbreviations used for convenience. Some examples:

Abbr.	Meaning	Notes
$r^+$	$(rr^*)$	1 or more occurrences
$r^?$	$(r \mid \epsilon)$	0 or 1 occurrence
$[a-z]$	$(a b \dots z)$	1 character in given range
$[abxyz]$	$(a b x y z)$	1 of the given characters

# Abbreviations

- Many systems allow abbreviations to make writing and reading definitions or specifications easier

name ::= *re*

Such a definition is not a valid RE.

A ::= B  
B ::= A

Finite-agent machines (FA) cannot detect this infinite loop.

- Restriction: abbreviations must not be circular (recursive) either directly or indirectly (else would be non-regular)

# Example

- Possible syntax for numeric constants

*digit* ::= [0-9]

*digits* ::= *digit*+

*number* ::= *digits* ( . *digits* )?

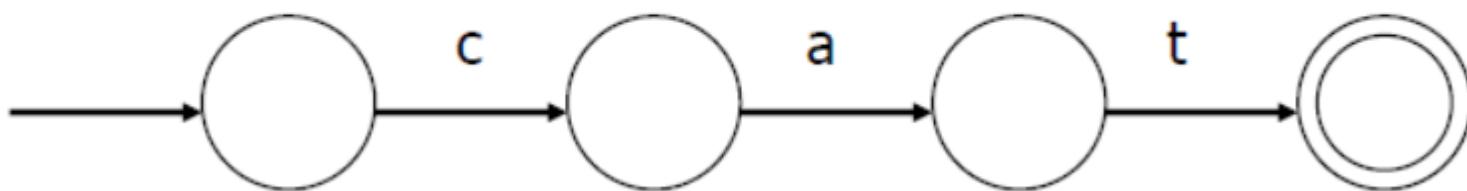
( [eE] (+ | -)? *digits* ) ?

- How would you describe this set in English?
- What are some examples of legal constants (strings) generated by *number* ?
  - What are the differences between these and numeric constants in YFPL? (Your Favorite Programming Language)

# Recognizing REs

- Finite automata can be used to recognize strings generated by regular expressions
- Can build by hand or automatically
  - Reasonably straightforward, and can be done systematically
  - Tools like Lex, Flex, JFlex et al do this automatically, given a set of REs
  - Same techniques used in grep, sed, other regular expression packages/tools

# Example: FSA for “cat”



String ?

# DFA vs NFA

- Deterministic Finite Automata (DFA)
  - No choice of which transition to take under any condition
  - No  $\epsilon$  transitions (arcs)
- Non-deterministic Finite Automata (NFA)
  - Choice of transition in at least one case
  - Accept if some way to reach a final state on given input
  - Reject if no possible way to final state
  - i.e., may need to guess right path or backtrack

# FAs in Scanners

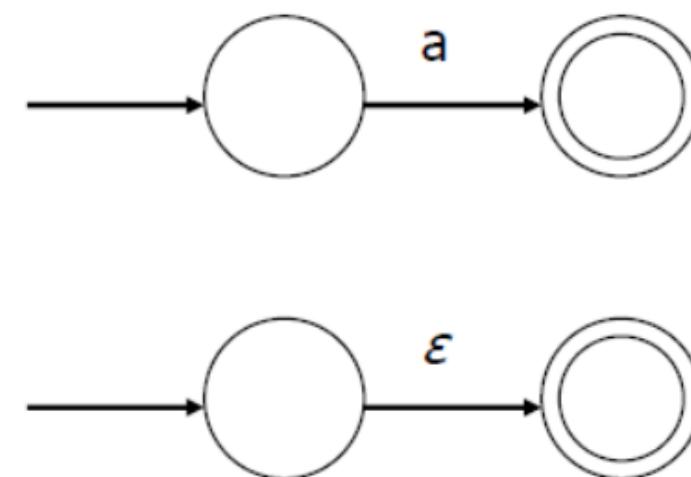
- Want DFA for speed (no backtracking)
- But conversion from regular expressions to NFA is easy
- Fortunately, there is a well-defined procedure for converting a NFA to an equivalent DFA (subset construction)

Constructing a DFA directly from a Regular Expression is sometimes difficult, while constructing an NFA from a Regular Expression is much easier.

In NFA?

# From RE to NFA: base cases

The first case shows that if the regular expression contains a special letter such as "a", the corresponding non-deterministic finite state machine moves from the start state to the accept state by a transition on the letter "a".



The second case has a transition on the symbol  $\epsilon$  (epsilon), which means a transition without reading any input. This means that the machine can go directly from the start state to the accept state without consuming any characters.

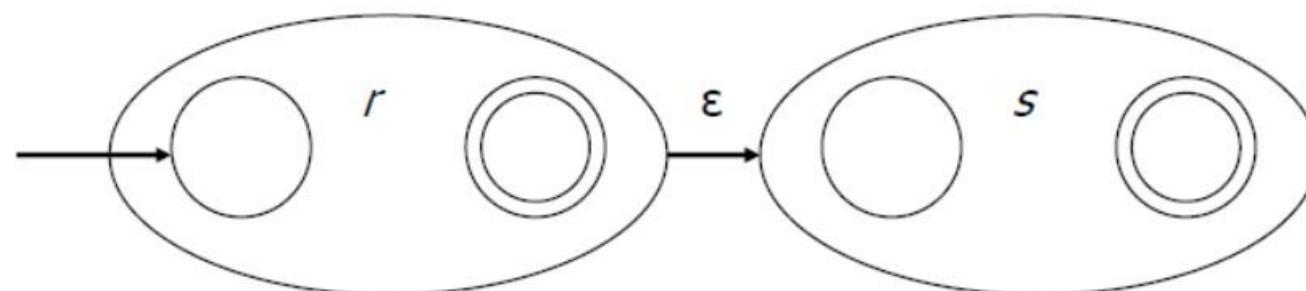
*r s*

String with Regular Expression

In NFA?

Transition?

Concatenation



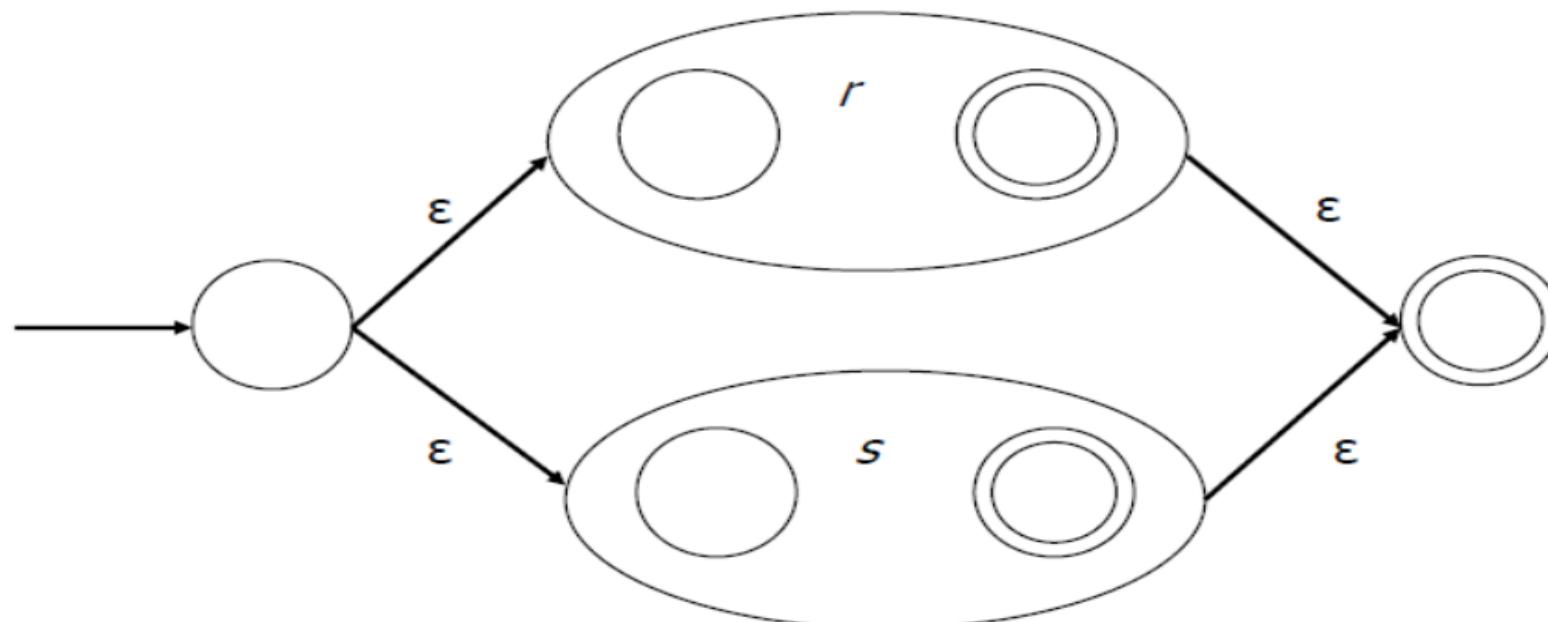
- The left-hand side represents an NFA that recognizes the regular expression  $r$ , consisting of an initial state and an accepting (final) state.
- The right-hand side shows an NFA for the regular expression  $s$ , also with its own initial and final states.
- There is an epsilon ( $\epsilon$ ) transition from the accepting state of  $r$ 's NFA directly to the start state of  $s$ 's NFA.

*r* | *s*

String with Regular Expression

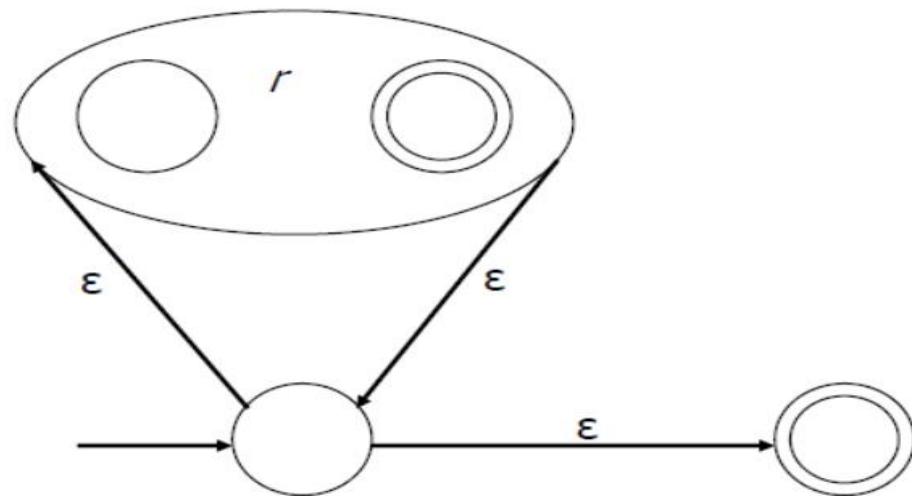
In NFA?

Transition?



- Create a new start state.
- Add  $\epsilon$ -transitions (epsilon transitions) from the new start state to the start states of the NFAs for *r* and *s*.
- The final states of the NFA for *r* and the NFA for *s* become the final states of the new NFA.

$r^*$



String with Regular Expression

In NFA?

Transition?

“Zero or m This NFA represents the Kleene star operation applied to a regular expression r.

The expression  $r^*$  means:

In terms of an NFA (Nondeterministic Finite Automaton), this means the machine accepts any string that can be made by concatenating zero or more strings that are accepted by r.

# Exercise

تمرین ۷

- Draw the NFA for:  $b(at|ag) \mid bug$

مهلت تحويل: ۲ ابان ماه ۱۴۰۴

موفق باشد



# به نام دانای نیآموخته

درس: کامپایلر

ترم: نیمسال اول (۰۵-۰۴)

دپارتمان: مهندسی کامپیووتر

مدرس: دکتر شیما شفیعی

# To Tokens

- A scanner is a DFA that finds the next token each time it is called
- Every “final” state of scanner DFA emits (returns) a token
- Tokens are the internal compiler names for the lexemes
  - == becomes EQUAL
  - ( becomes LPAREN
  - while becomes WHILE
  - xyzzy becomes ID(xyzzy)
- You choose the names
- Also, there may be additional data ... \r\n might count lines; token data structure might include source line numbers

## DFA => Code

- **Option 1: Implement by hand using procedures**
  - one procedure for each token
  - each procedure reads one character
  - choices implemented using if and switch statements
- **Pros**
  - straightforward to write
  - fast
- **Cons**
  - a lot of tedious work
  - may have subtle differences from the language specification

## DFA => Code [continued]

- Option 1a: Like option 1, but structured as a single procedure with multiple return points
  - choices implemented using if and switch statements
- Pros
  - also straightforward to write
  - faster
- Cons
  - a lot of tedious work
  - may have subtle differences from the language specification

# DFA => code [continued]

- Option 2: use tool to generate table driven scanner
  - Rows: states of DFA
  - Columns: input characters
  - Entries: action
    - Go to next state
    - Accept token, go to start state
    - Error
- Pros
  - Convenient
  - Exactly matches specification, if tool generated
- Cons
  - “Magic”

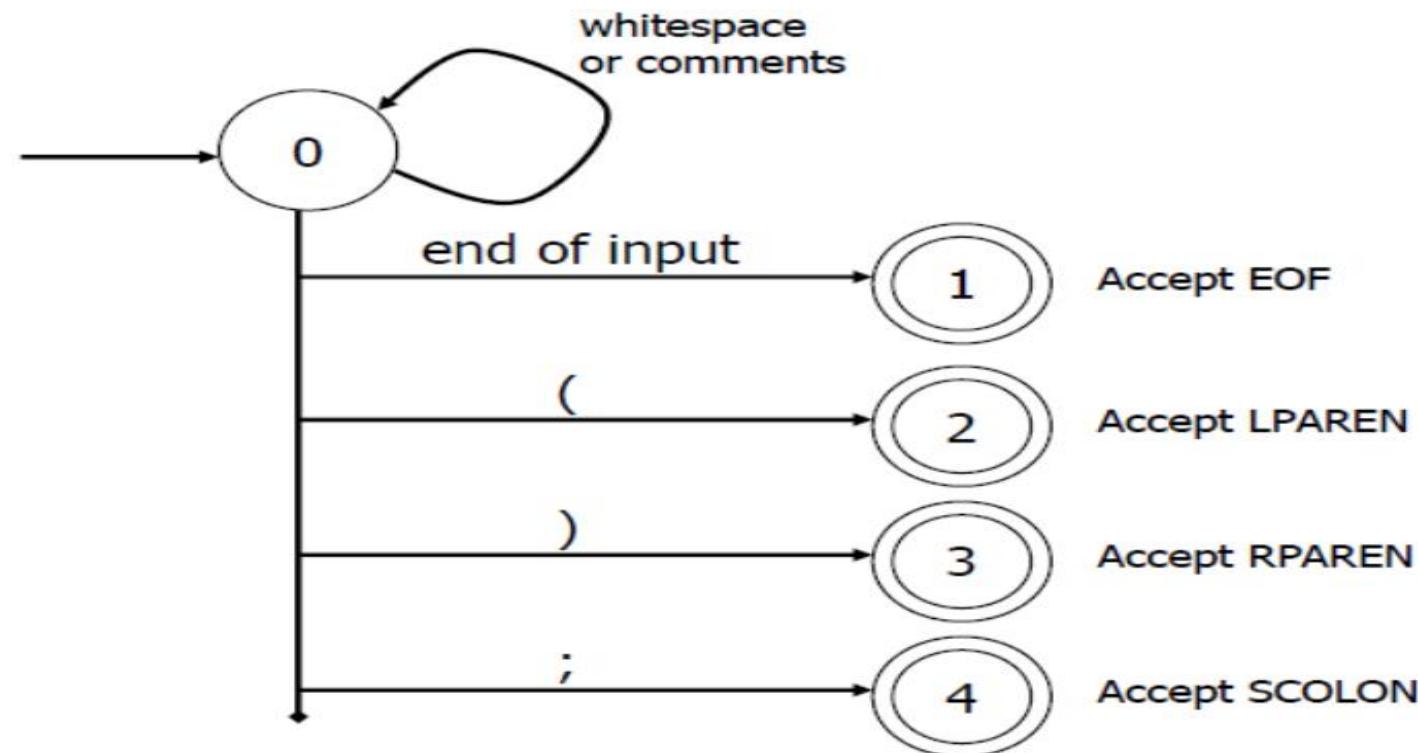
# DFA => code [continued]

- Option 2a: use tool to generate scanner
  - Transitions embedded in the code
  - Choices use conditional statements, loops
- Pros
  - Convenient
  - Exactly matches specification, if tool generated
- Cons
  - “Magic”
  - Lots of code – big but potentially quite fast
    - Would never write something like this by hand, but can generate it easily enough

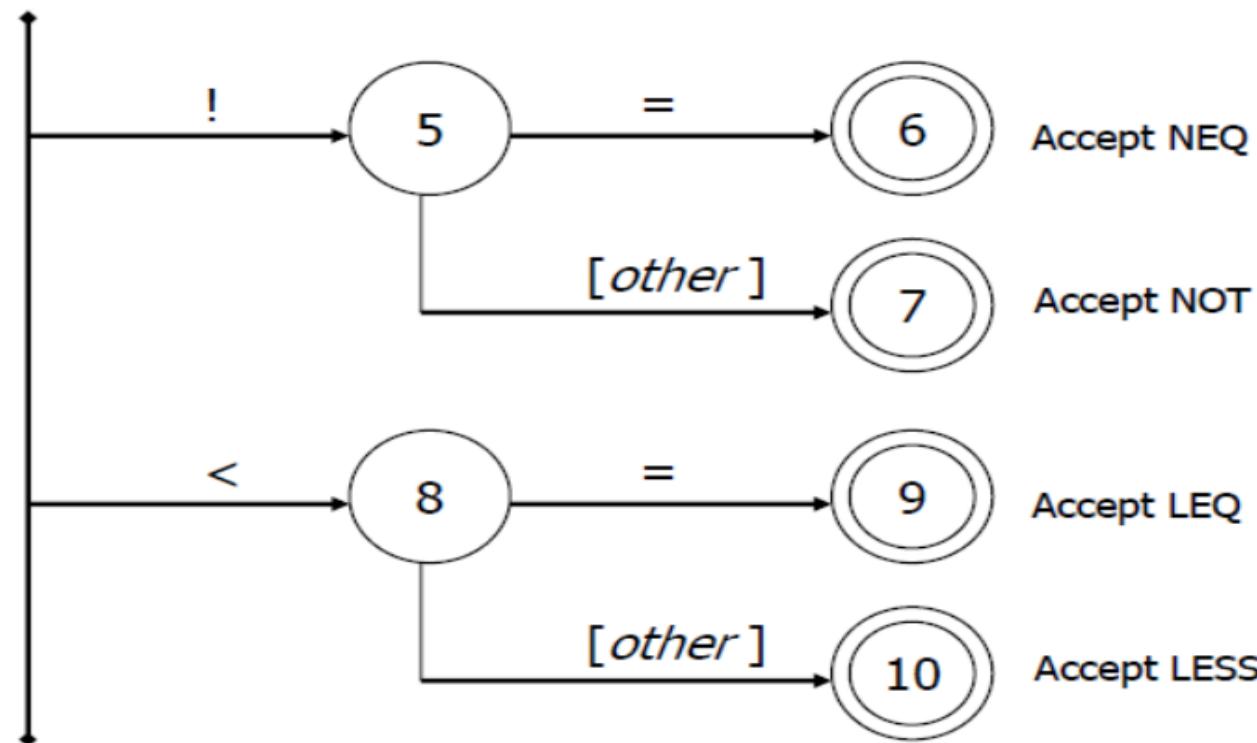
## Example: DFA for hand-written scanner

- Idea: show a hand-written DFA for some typical programming language constructs
  - Then use to outline a hand-written scanner
- Setting: Scanner is called when the parser needs a new token
  - Scanner knows (saves) current position in input
  - From there, use a DFA to recognize the longest possible input sequence that makes up a token and return that token; save updated position for next time
- Disclaimer: Example for illustration only – you'll use tools for the course project
  - & we're abusing the DFA notation a little – not all arrows in the diagram correspond to consuming an input character, but meaning should be pretty obvious

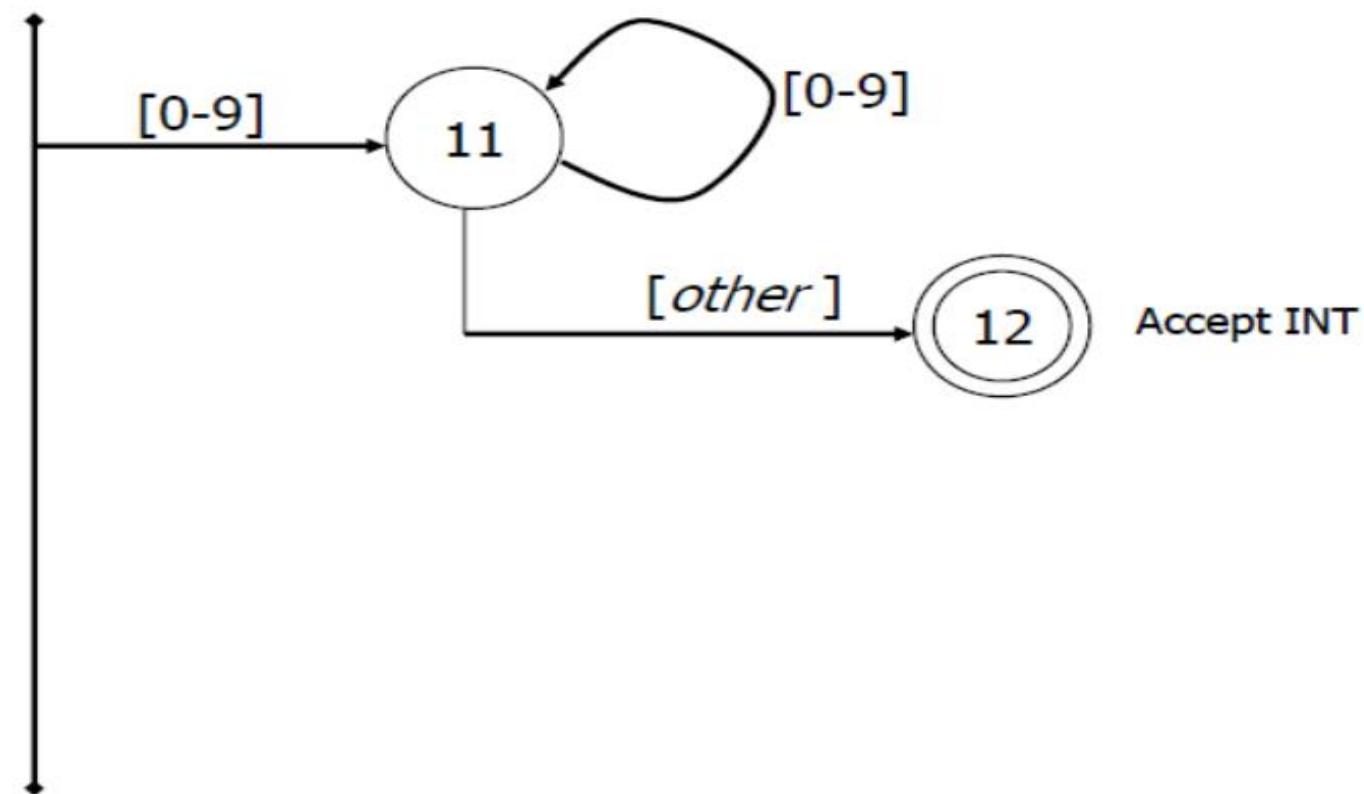
# Scanner DFA Example (1)



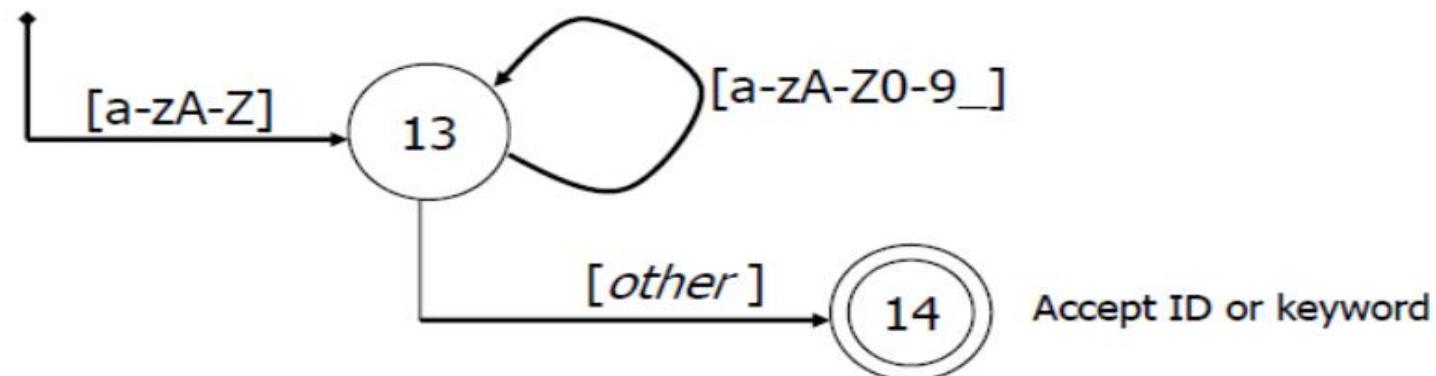
# Scanner DFA Example (2)



# Scanner DFA Example (3)



## Scanner DFA Example (4)



- Strategies for handling identifiers vs keywords
  - Hand-written scanner: look up identifier-like things in table of keywords to classify (good application of perfect hashing)
  - Machine-generated scanner: generate DFA with appropriate transitions to recognize keywords
    - Lots 'o states, but efficient (no extra lookup step)

# Implementing a Scanner by Hand – Token Representation

- A token is a simple, tagged structure

```
public class Token {  
    public int kind;          // token's lexical class  
    public int intval;        // integer value if class = INT  
    public String id;         // actual identifier if class = ID  
    // useful extra information for debugging / diagnostics:  
    public int line;  
    public int column;  
    // lexical classes (ancient java – better to use enums)  
    public static final int EOF = 0;    // "end of file" token  
    public static final int ID  = 1;    // identifier, not keyword  
    public static final int INT = 2;    // integer  
    public static final int LPAREN = 4;  
    public static final int SCOLN = 5;  
    public static final int WHILE = 6;  
    // etc. etc. etc. ...
```

# Scanner getToken() method

```
// return next input token
public Token getToken() {
    Token result;

    skipWhiteSpace();

    if (no more input) {
        result = new Token(Token.EOF); return result;
    }

    switch(nextch) {
        case '(': result = new Token(Token.LPAREN); getch(); return result;
        case ')': result = new Token(Token.RPAREN); getch(); return result;
        case ';': result = new Token(Token.SCOLON); getch(); return result;

        // etc. ...
    }
}
```

## getToken() (2)

```
case '!': // ! or !=  
    getch();  
    if (nextch == '=') {  
        result = new Token(Token.NEQ); getch(); return result;  
    } else {  
        result = new Token(Token.NOT); return result;  
    }  
  
case '<': // < or <=  
    getch();  
    if (nextch == '=') {  
        result = new Token(Token.LEQ); getch(); return result;  
    } else {  
        result = new Token(Token.LESS); return result;  
    }  
// etc. ...
```

## تمرین ۸

Explain what this code fragment does.

Describe the purpose of each of the following statements:

getch();

while (nextch is a digit)

new Token(Token.INT, Integer(num).intValue());

```
case '0': case '1': case '2': case '3': case '4':  
case '5': case '6': case '7': case '8': case '9':  
    // integer constant  
    String num = nextch;  
    getch();  
    while (nextch is a digit) {  
        num = num + nextch; getch();  
    }  
    result = new Token(Token.INT, Integer(num).intValue());  
    return result;
```

مهلت تحويل: ۹ آبان ۱۴۰۴

موفق باشید



# به نام دانای نیا موخته

درس: کامپایلر

ترم: نیمسال اول (۰۵-۰۴)

دپارتمان: مهندسی کامپیوتر

مدرس: دکتر شیما شفیعی

## Syntactic Analysis / Parsing

- Goal: Convert the token stream to an abstract syntax tree
- Abstract syntax tree (AST):
  - Captures the structural features of the program
  - Primary data structure for the next phases of compilation
- Plan
  - Study how context-free grammars specify syntax
  - Study algorithms for parsing and building ASTs

## Context-free Grammars

- The syntax of most programming languages can be specified by a context-free grammar (CFG)
- Compromise between
  - REs: can't nest or specify recursive structure
  - General grammars: too powerful, undecidable
- Context-free grammars are a sweet spot
  - Powerful enough to describe nesting, recursion
  - Easy to parse, but also allows restrictions for speed
- Not perfect
  - Cannot capture semantics, like “must declare every variable” or “must be `int`” – requires later semantic pass
  - Can be ambiguous

## Derivations and Parse Trees

- Derivation: a sequence of expansion steps, beginning with a start symbol and leading to a sequence of terminals
- Parsing: inverse of derivation
  - Given a sequence of terminals (aka tokens) want to recover (discover) the nonterminals and structure, i.e., the parse (concrete syntax) tree

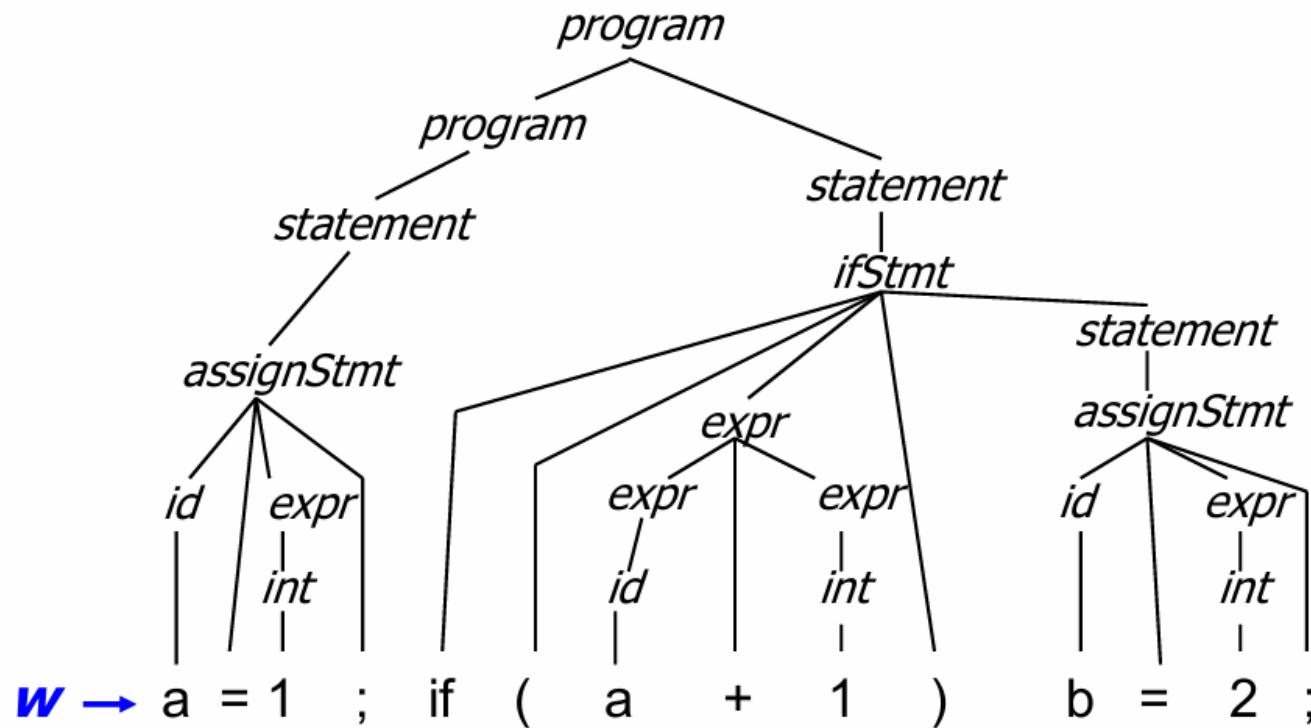
# Old Example

*G*

```

program ::= statement | program statement
statement ::= assignStmt | ifStmt
assignStmt ::= id = expr ;
ifStmt ::= if ( expr ) statement
expr ::= id | int | expr + expr
id ::= a | b | c | i | j | k | n | x | y | z
int ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

```



Feature	Top-Down Parsing	Bottom-Up Parsing
Tree construction direction	From root to leaves	From leaves to roots
Starting point	Start symbol (e.g., program)	Input string
Example algorithms	Recursive Descent, LL(1)	LR(0), SLR, LALR

# Parsing

- Parsing: Given a grammar  $G$  and a sentence  $w$  in  $L(G)$ , traverse the derivation (parse tree) for  $w$  in some *standard order* and do *something useful* at each node
  - The tree might not be produced explicitly, but the control flow of the parser will correspond to a traversal

Explanation	Concept
The process of analyzing a sentence according to the grammar rules	Parsing
To build or traverse the parse tree for the sentence	Goal
The parse tree might not be built explicitly, but the parser's control flow corresponds to its traversal	Note

# "Standard Order"

- For practical reasons we want the parser to be *deterministic* (no backtracking), and we want to examine the source program from *left to right*.
  - (i.e., parse the program in linear time in the order it appears in the source file)

"Standard order" means having a parser that parses the source code from left to right in linear time, without guessing or recursion.

# Common Orderings

- Top-down
  - Start with the root
  - Traverse the parse tree depth-first, left-to-right (leftmost derivation)
  - LL(k), recursive-descent
- Bottom-up
  - Start at leaves and build up to the root
    - Effectively a rightmost derivation in reverse(!)
  - LR(k) and subsets (LALR(k), SLR(k), etc.)

Parsing Method	Direction / Strategy	Derivation Type	Typical Parsers
Top-Down Parsing	Start from the root; traverse depth-first, left-to-right	Leftmost derivation	LL(k), Recursive-Descent
Bottom-Up Parsing	Start from the leaves; build up to the root	Rightmost derivation in reverse	LR(k), LALR(k), SLR(k)

### Top-down Parsing (LL(k), Recursive Descent):

Consider the grammar:

$$S \rightarrow aSb \mid ab$$

Input string: *aabb*

- Start from root  $S$ .
- Parse leftmost first:
  - $S \Rightarrow aSb$
  - $S \Rightarrow ab$
- Trace:
  - $S \Rightarrow aSb \Rightarrow aabb$
- This is a depth-first left-to-right traversal, representing a leftmost derivation.

### Bottom-up Parsing (LR(k)):

Using the same grammar and input *aabb*:

- Start at leaves *a, a, b, b*
- Combine adjacent symbols and go upward:
  - Reduce *ab* to  $S$
  - Then reduce  $aSb$  to  $S$
- This corresponds to a rightmost derivation in reverse order.
- Parser uses shift and reduce to build the parse tree from leaves to root.

# “Something Useful”

- At each point (node) in the traversal, perform some semantic action
  - Construct nodes of full parse tree (rare)
  - Construct abstract syntax tree (AST) (common)
  - Construct linear, lower-level representation (often produced by traversing initial AST in later phases of production compilers)
  - Generate target code on the fly (used in 1-pass compiler; not common in production compilers)
    - Can't generate great code in one pass, but useful if you need a quick 'n dirty working compiler

# EX

a := b + 1;

assign → ident := expr

expr → expr + term | term

term → ident | number

## Reduced Grammars

- Grammar  $G$  is *reduced* iff for every production  $A ::= \alpha$  in  $G$  there is a derivation
$$S \Rightarrow^* x A z \Rightarrow x \alpha z \Rightarrow^* xyz$$
– i.e., no production is useless
- Convention: we will use only reduced grammars
  - There are algorithms for pruning useless productions from grammars – see a formal language or compiler book for details

# EX

$$\begin{aligned}S &\rightarrow AB \\A &\rightarrow a \\B &\rightarrow bB \mid b \\C &\rightarrow c\end{aligned}$$

Here, the nonterminal  $C$  and the production  $C \rightarrow c$  are useless because  $C$  cannot be reached from the start symbol  $S$ , so they do not produce any string in the language.

After removing the useless productions involving  $C$ , the grammar becomes reduced:

$$\begin{aligned}S &\rightarrow AB \\A &\rightarrow a \\B &\rightarrow bB \mid b\end{aligned}$$

## تمرین ۹

```
expr ::= expr + expr | expr - expr  
       | expr * expr | expr / expr | int  
int ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

- Give a different leftmost derivation of  $2+3*4$  and show the parse tree

مهلت تحويل: ۹ آبان ۱۴۰۴

موفق باشد



# به نام دانای نیآموخته

درس: کامپیالر

ترم: نیمسال اول (۰۵-۰۴)

دپارتمان: مهندسی کامپیووتر

مدرس: دکتر شیما شفیعی

# Ambiguity

- Grammar  $G$  is *unambiguous* iff every  $w$  in  $L(G)$  has a unique leftmost (or rightmost) derivation
  - Fact: unique leftmost or unique rightmost implies the other
- A grammar without this property is *ambiguous*
  - Note that other grammars that generate the same language may be unambiguous
- We need unambiguous grammars for parsing

They enable unique and reliable parsing; each string has exactly one **derivation tree**.

**Parsing:** Ambiguity causes confusion and errors in the translation and execution of programs written in that grammar language.

An example of leftmost derivation for the expression  $a + (b * c)$  is:

$$E \Rightarrow E + E$$

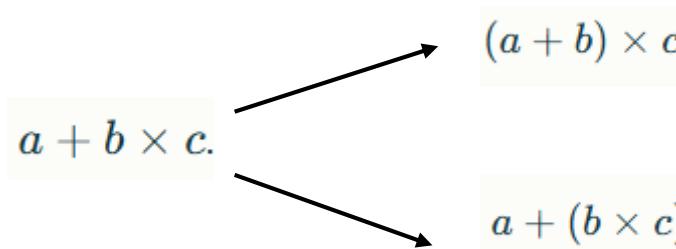
$$E \Rightarrow a + E$$

$$E \Rightarrow a + (E)$$

$$E \Rightarrow a + (E * E)$$

$$E \Rightarrow a + (b * E)$$

$$E \Rightarrow a + (b * c)$$



## Example: Ambiguous Grammar for Arithmetic Expressions

*expr ::= expr + expr | expr - expr  
| expr \* expr | expr / expr | int*

*int ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9*

- Exercise: show that this is ambiguous
  - How? Show two different leftmost or rightmost derivations for the same string
  - Equivalently: show two different parse trees for the same string

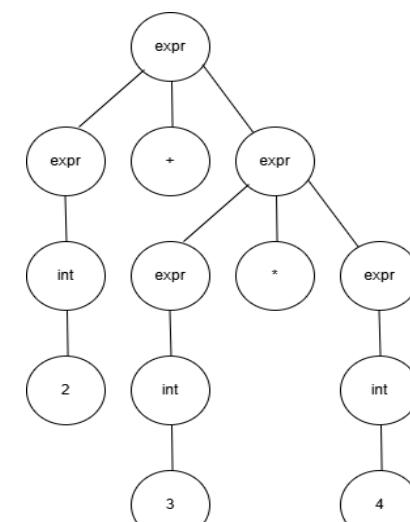
## Example (cont)

$$\begin{aligned} \text{expr} ::= & \text{expr} + \text{expr} \mid \text{expr} - \text{expr} \\ & \mid \text{expr} * \text{expr} \mid \text{expr} / \text{expr} \mid \text{int} \\ \text{int} ::= & 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \end{aligned}$$

Ambiguous

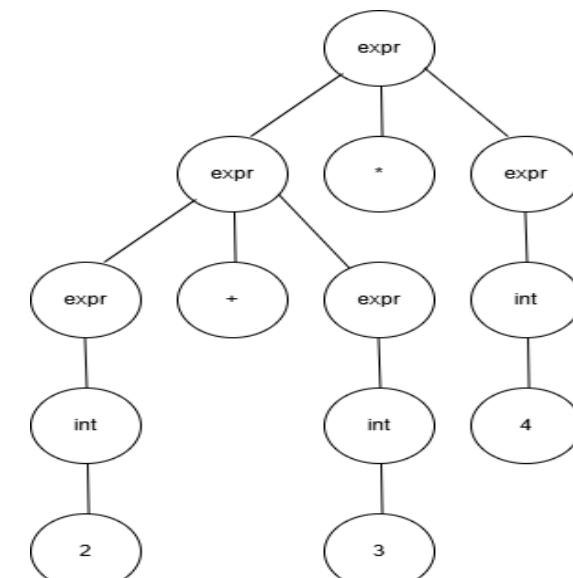
- Give a leftmost derivation of  $2+3*4$  and show the parse tree

Leftmost derivation means: At each stage of derivation, we always choose the leftmost non-terminal symbol to expand (generate).



Different parse trees for the same string

(Leftmost)



(Rightmost)

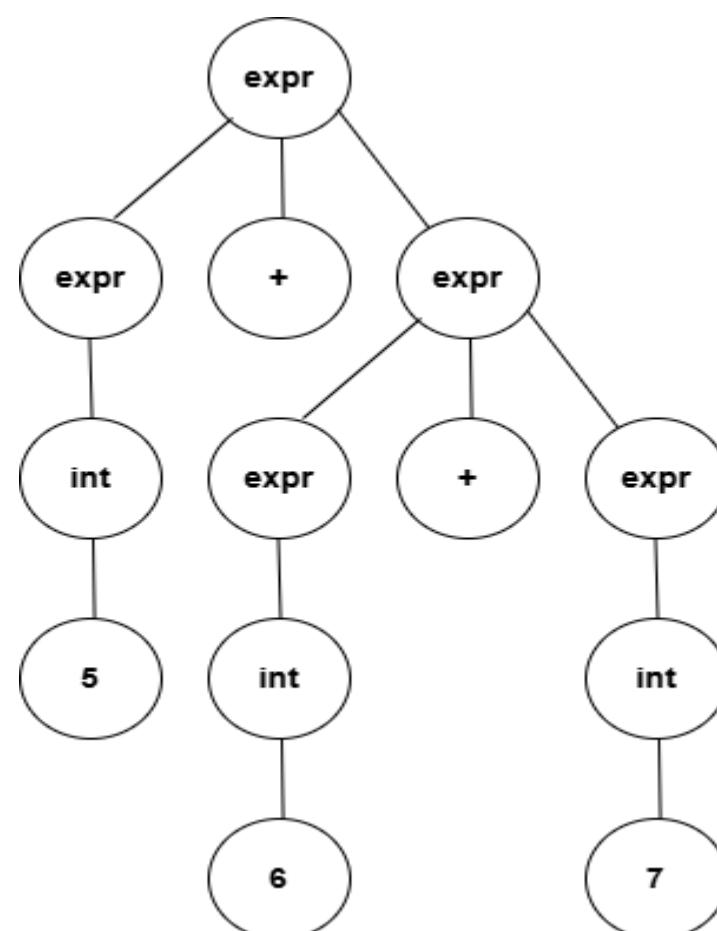
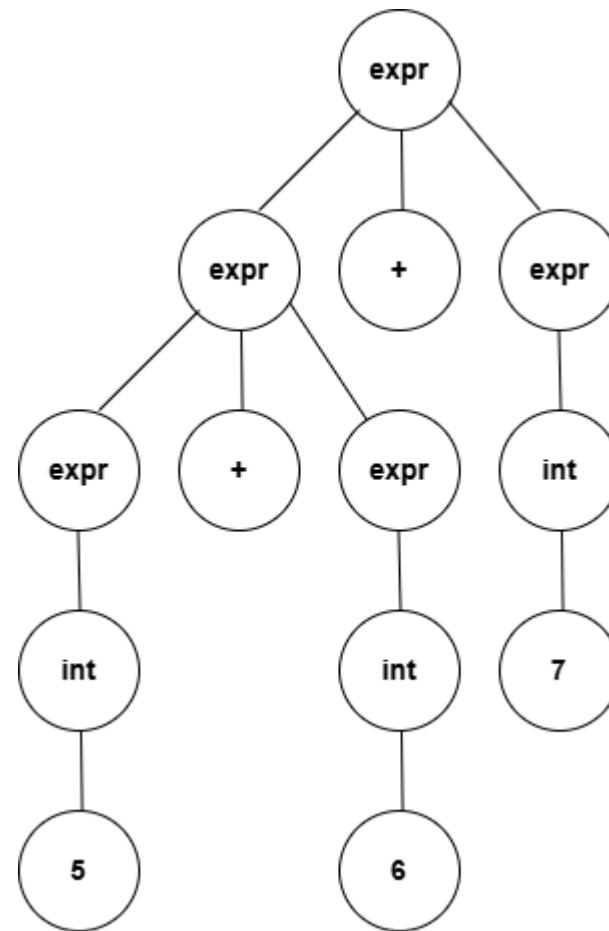
```

expr ::= expr + expr | expr - expr
      | expr * expr | expr / expr | int
int ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

```

## Example (cont)

- Give a leftmost derivation of  $2+3*4$  and show the parse tree



Ambiguous

# What's going on here?

- The grammar has no notion of precedence or associativity
- Traditional solution
  - Create a non-terminal for each level of precedence
  - Isolate the corresponding part of the grammar
  - Force the parser to recognize higher precedence subexpressions first
  - Use left- or right-recursion for left- or right-associative operators (non-associative operators are not recursive)

Goal	Method
Define operator precedence	Create a separate non-terminal for each level of precedence
Ensure correct order of evaluation	Use a hierarchical grammar structure ( $\text{Expr} \rightarrow \text{Term} \rightarrow \text{Factor}$ )
Control operator associativity	Use left or right recursion (left for left-associative, right for right-associative operators)

## Left Recursion

a - b - c

(a - b) - c

a - (b - c)



+  
-  
(

## Right Recursion

<power>  $\rightarrow$  <factor>  $\wedge$  <power> | <factor>

a  $\wedge$  b  $\wedge$  c

(b  $\wedge$  c)

$\wedge$

# Classic Expression Grammar

(first used in ALGOL 60)

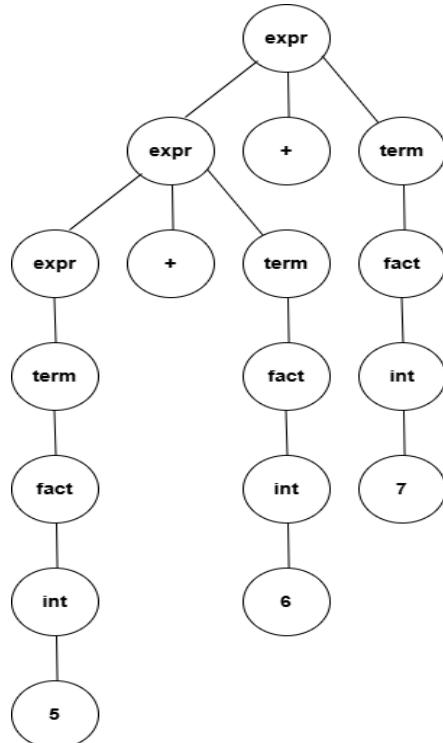
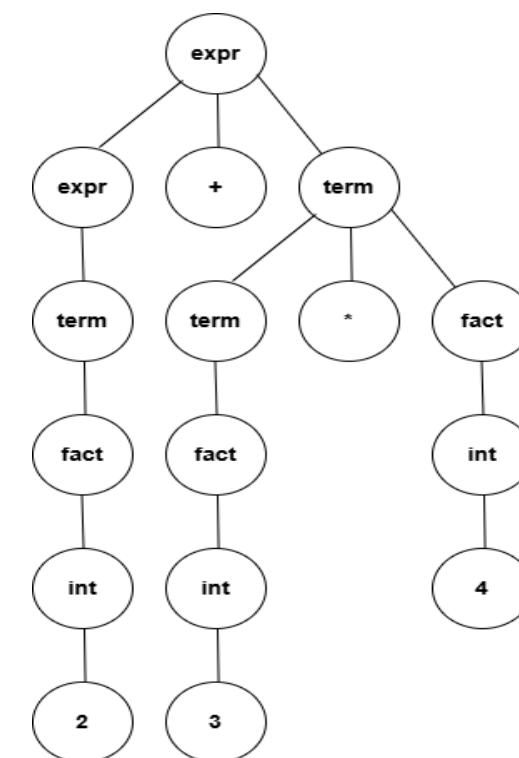
$expr ::= expr + term \mid expr - term \mid term$

$term ::= term * factor \mid term / factor \mid factor$

$factor ::= int \mid ( expr )$

$int ::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7$

Example	Expression	Ambiguity	Explanation
(1) Left side	$5 + 6 + 7$	Ambiguous	Because it allows both $((5 + 6) + 7)$ and $(5 + (6 + 7))$
(2) Right side	$2 + 3 * 4$	Unambiguous	Because grammar enforces precedence (* before +)



Level	Non-terminal	Typical Operators	Example	Precedence
1 Lowest	expr	+, -	$2 + 3 * 4$	Lowest
2 Middle	term	*, /	$3 * 4$	Higher
3 Highest	factor	none (just int or (expr))	$(4 + 5) \text{ or } 7$	Highest

Classic Expression Grammar is a grammar that was first introduced in ALGOL 60

One of the first programming languages to use formal grammars (BNF).

The expression grammar in ALGOL 60 was designed to:

Operator precedence is naturally respected

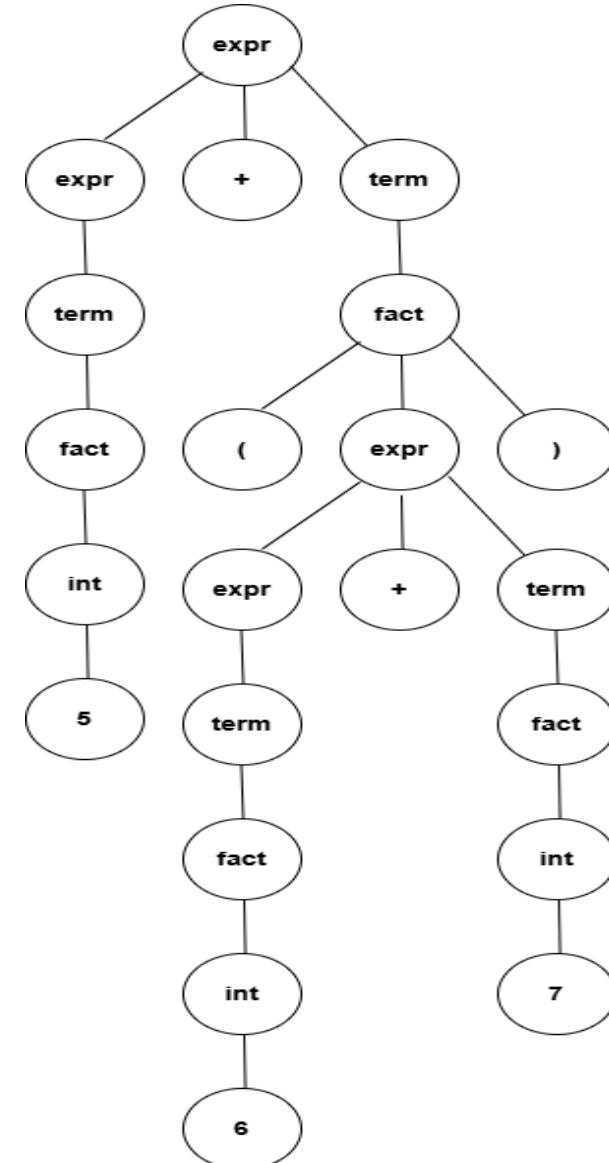
Associativity is also determined through left/right recursion

Check:

Derive  $5 + (6 + 7)$

```

expr ::= expr + term | expr - term | term
term ::= term * factor | term / factor | factor
factor ::= int | ( expr )
int ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7
  
```



- This grammar is disambiguated because:
- It defines each level of operators separately in a non-terminal
- It uses the left return for left-direction
- It uses parentheses to manually control precedence

# Another Classic Example

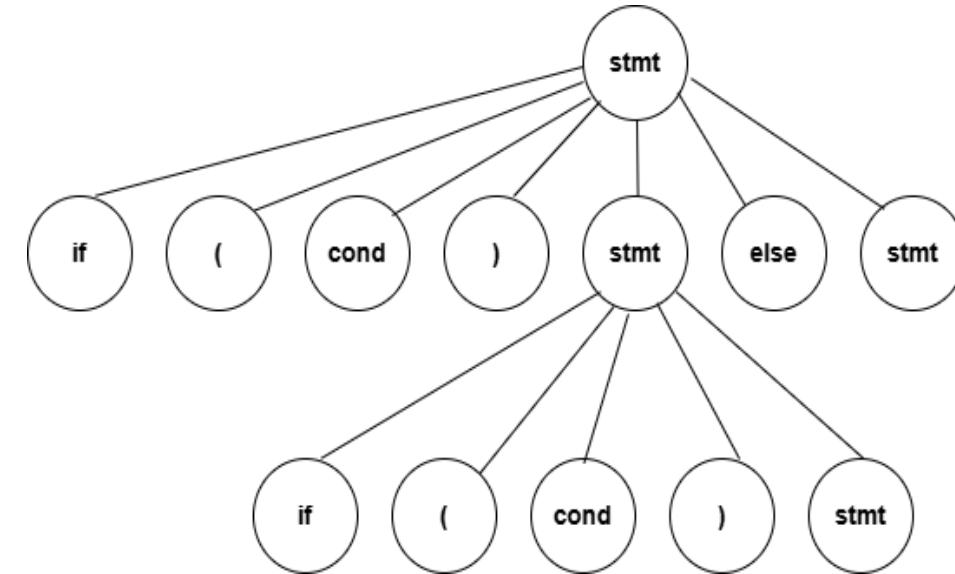
- Grammar for conditional statements

```
stmt ::= if ( cond ) stmt  
      | if ( cond ) stmt else stmt
```

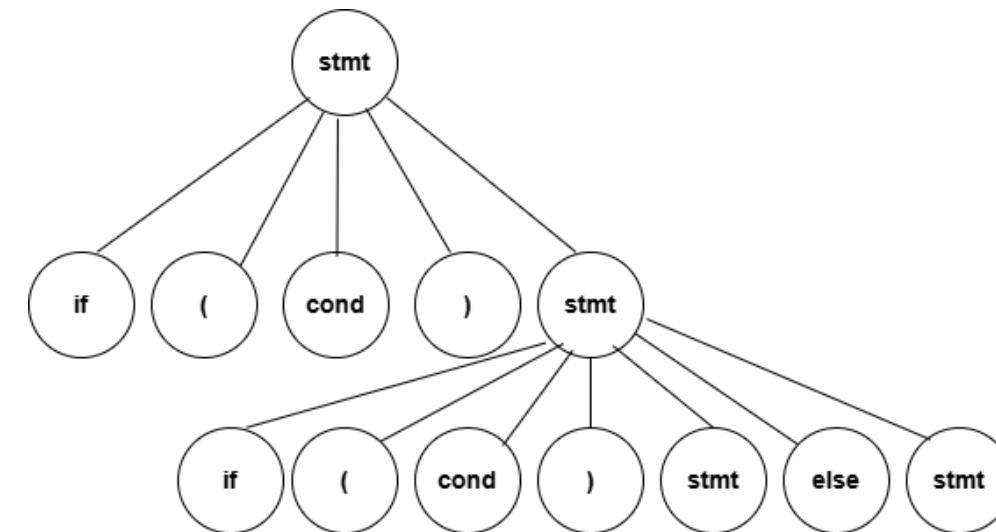
- Exercise: show that this is ambiguous
  - How?

## One Derivation

$stmt ::= if ( cond ) stmt$   
|  $if ( cond ) stmt else stmt$



`if ( cond ) if ( cond ) stmt else stmt`



# Solving “if” Ambiguity

- Fix the grammar to separate if statements with else clause and if statements with no else
  - Done in Java reference grammar
  - Adds lots of non-terminals
- or, Change the language
  - But it'd better be ok to do this
- or, Use some ad-hoc rule in the parser
  - “else matches closest unpaired if”

# تمرین ۱

```
stmt ::= if ( cond ) stmt  
      | if ( cond ) stmt else stmt
```

- `if ( cond ) if ( cond ) if (cond) stmt else stmt`
- Ambiguous or non-Ambiguous? Why?



# به نام دانای نیآموخته

درس: کامپایلر

ترم: نیمسال اول (۰۵-۰۴)

دپارتمان: مهندسی کامپیووتر

مدرس: دکتر شیما شفیعی

# Solving “if” Ambiguity

- Fix the grammar to separate if statements with else clause and if statements with no else
  - Done in Java reference grammar
  - Adds lots of non-terminals
- or, Change the language
  - But it'd better be ok to do this
- or, Use some ad-hoc rule in the parser
  - “else matches closest unpaired if”

"dangling else problem"

### Leftmost derivation:

1.  $E \Rightarrow E + E$
2.  $E + E \Rightarrow id + E$
3.  $id + E \Rightarrow id + E * E$
4.  $id + E * E \Rightarrow id + id * E$
5.  $id + id * E \Rightarrow id + id * id$

For example, consider the grammar:

$$E \rightarrow E + E \mid E * E \mid id$$

and the input string:

$$id + id * id.$$

### Rightmost derivation:

1.  $E \Rightarrow E + E$
2.  $E + E \Rightarrow E + E * E$
3.  $E + E * E \Rightarrow E + E * id$
4.  $E + E * id \Rightarrow E + id * id$
5.  $E + id * id \Rightarrow id + id * id$

## Resolving Ambiguity with Grammar (2)

- If you can (re-)design the language, just avoid the problem entirely

Stmt ::= ... |

**if Expr then Stmt end** |

**if Expr then Stmt else Stmt end**

getValue()

print("Hello")

- formal, clear, elegant
- allows sequence of Stmts in then and else branches, no { , } needed
- extra end required for every if  
(But maybe this is a good idea anyway?)

# Parser Tools and Operators

- Most parser tools can cope with ambiguous grammars
  - Makes life simpler if used with discipline
- Usually can specify precedence & associativity
  - Allows simpler, ambiguous grammar with fewer nonterminals as basis for parser – let the tool handle the details (but only when it makes sense)
    - (i.e.,  $\text{expr} ::= \text{expr} + \text{expr} \mid \text{expr}^* \text{expr} \mid \dots$  with assoc. & precedence declarations can be the best solution)
- Take advantage of this to simplify the grammar when using parser-generator tools

# Parser Tools and Ambiguous Grammars

- Possible rules for resolving other problems
  - Earlier productions in the grammar preferred to later ones (danger here if parser input changes)
  - Longest match used if there is a choice (good solution for dangling if)
- Parser tools normally allow for this
  - But be sure that what the tool does is really what you want
    - And that it's part of the tool spec, so that v2 won't do something different (that you *don't* want!)

## Ex

## Parser کاربردی ابزارهای

Operator	Precedence	Associativity	Example	Explanation
*	2	Left	$3 + 4 * 5$	Multiplication has higher precedence; compute $4 * 5$ first, then add 3.
+	1	Left	$3 + 4 + 5$	Left associativity means $(3 + 4) + 5$ .
$\wedge$	3	Right	$2 \wedge 3 \wedge 2$	Right associativity means $2 \wedge (3 \wedge 2)$

# Type of parsing method

Parser Type	Parsing Method	Typical Grammar	Characteristics
LL(k)	Top-down	Grammars without left recursion	Simple, fast, limited

	Step	Action Description
E ::= id   id + E	1	LL(1) looks at one lookahead token to decide production rule.
( E )	2	Observes (, applies rule S ::= ( E ).
LL(k=1)?	3	Next, need to parse E. With lookahead id, chooses `E ::= id + E
	4	Matches id terminal from input.
	5	Sees +, continues parsing id + E.
	6	Matches next id and then ).
	7	Parsing completes successfully.

Parser Type	Parsing Method	Typical Grammar	Characteristics
LR(k)	Bottom-up	Grammars with left recursion	Powerful, can handle complex grammars

E ::= E + id   id	Step	Action Description
id + id + id LR(k=1)?	1	See id, reduce to E.
	2	Read + and id next.
	3	Reduce E + id to E.
	4	Read next + and id.
	5	Reduce E + id to E again.
	6	Input fully parsed, stack contains E. Parse successful.

Shift: Read a new symbol from the input and push it onto the stack (Last in-First out).

Reduce: Replace a combination of symbols on the stack with a non-terminal symbol based on a grammar rule.

$S \rightarrow CC$

$C \rightarrow aC \mid d$

a a d d

- Shift means reading a symbol and pushing the corresponding state on the stack.
- Reduce means replacing the right-hand side symbols matching a production with its left-hand nonterminal.
- This bottom-up process continues until the input is fully parsed and reduced to the start symbol S.

Step	Stack	Input	Action
1	0	a a d d	Shift a (to state 3)
2	0 3	a d d	Shift a (to state 3)
3	0 3 3	d d	Shift d (to state 4)
4	0 3 3 4	d	Reduce d to C
5	0 3 3	d	Reduce aC to C
6	0 3	d	Shift d (to state 4)
7	0 3 4		Reduce d to C
8	0 3		Reduce aC to C
9	0		Reduce CC to S
10			Accept

## تمرین ۱۱

1. Show two different parse trees for this string that demonstrate the ambiguity of the grammar.
2. Explain why this grammar is ambiguous.

$$E \rightarrow E + E \mid E * E \mid (E) \mid id$$

مهلت تحويل: ۱۶ آبان ۱۴۰۴

موفق باشید



# به نام دانای نیآموخته

درس: کامپایلر

ترم: نیمسال اول (۰۵-۰۴)

دپارتمان: مهندسی کامپیووتر

مدرس: دکتر شیما شفیعی

# Bottom-Up Parsing

- Idea: Read the input left to right
- Whenever we've matched the right hand side of a production, reduce it to the appropriate non-terminal and add that non-terminal to the parse tree
- The upper edge of this partial parse tree is known as the *frontier*

# Example

- Grammar
- Bottom-up Parse

$S ::= aABe$

$A ::= Abc \mid b$

$B ::= d$

الگوی کلی رشته‌ها

$a (b^n) (c^{n-1}) d e , \quad n \geq 1$

Step	Stack	Remaining Input	Action
1	a	b d e	shift 'a'
2	a b	d e	shift 'b'
3	a b d	e	shift 'd'
4	a b B	e	reduce 'd' to B
5	a A	e	reduce 'b B' to A
6	a A e	-	shift 'e'
7	S	-	reduce 'a A e' to S

a b d e

# LR(1) Parsing

- We'll look at LR(1) parsers
  - Left to right scan, Rightmost derivation, 1 symbol lookahead
  - Almost all practical programming languages have an LR(1) grammar
  - LALR(1), SLR(1), etc. – subsets of LR(1)
    - LALR(1) can parse most real languages, tables are more compact, and is used by YACC/Bison/CUP/etc.

# LR Parsing in Greek

- The bottom-up parser reconstructs a reverse rightmost derivation
- Given the rightmost derivation

$$S \Rightarrow \beta_1 \Rightarrow \beta_2 \Rightarrow \dots \Rightarrow \beta_{n-2} \Rightarrow \beta_{n-1} \Rightarrow \beta_n = w$$

the parser will first discover  $\beta_{n-1} \Rightarrow \beta_n$  , then  $\beta_{n-2} \Rightarrow \beta_{n-1}$  , etc.

- Parsing terminates when
  - $\beta_1$  reduced to  $S$  (start symbol, success), or
  - No match can be found (syntax error)

## Grammar:

1.  $S \rightarrow aA$
2.  $A \rightarrow b$

Input String: “a c”

Parsing table of steps:

Step	Stack	Remaining Input	Action
1		a c	shift ‘a’
2	a	c	Error (syntax)

## Grammar:

1.  $S \rightarrow aA$
2.  $A \rightarrow b$

Input String: “a b”

Parsing table of steps:

Step	Stack	Remaining Input	Action
1		a b	shift ‘a’
2	a	b	shift ‘b’
3	a b		reduce $A \rightarrow b$
4	a A		reduce $S \rightarrow aA$
5	S		accept

## تمرین ۱۲

- Given the grammar:

- $S \rightarrow L = R$
- $S \rightarrow R$
- $L \rightarrow * R$
- $L \rightarrow id$
- $R \rightarrow L$

- And the input string:

$id = * id$

1. Using the shift-reduce parsing technique, show each step of parsing the input string.

2. Specify at each step:

- The current stack content
- Remaining input
- The action taken (shift, reduce by which rule, accept, or error)

مهلت تحويل: ۱۶ آبان ۱۴۰۴