Compiler Design

Fatemeh Deldar

Isfahan University of Technology

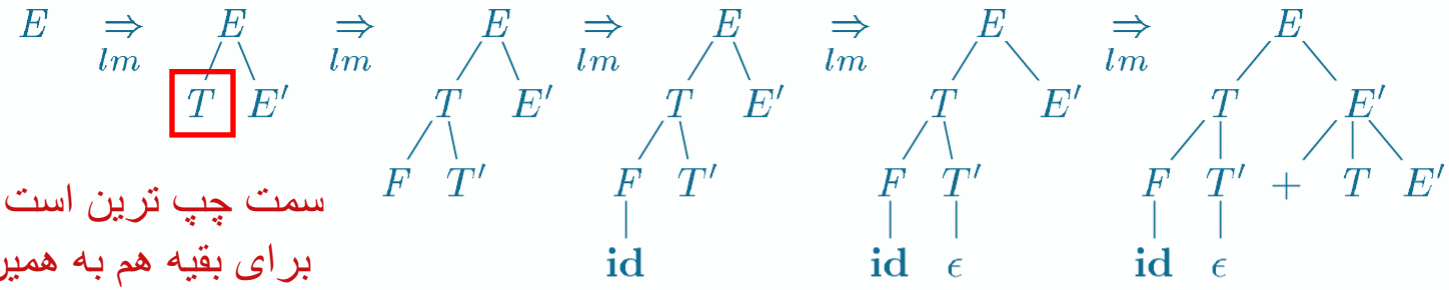
1402-1403

Top-Down Parsing

- Top-down parsing can be viewed as the problem of constructing a parse tree for the input string, starting from the root and creating the nodes of the parse tree in *preorder* (depth-first)
- Top-down parsing can be viewed as finding a *leftmost derivation* for an input string
- ***LL*(*k*) grammars**
  - The class of grammars for which we can construct predictive parsers looking k symbols ahead in the input

-
توی بالا به پایین اون درخت Parsing که ساخته میشه از ریشه شروع میشه به ساختن
این بالا به پایین پیمایشش به صورت preorder است
preorder ینی اول برای هر گره ای خود گره ایجاد میشه و بعد سمت چپش ایجاد میشه و بعد سمت راستش ایجاد میشه
این پارسرای بالا به پایین اشتقاقی که ازشون به دست میاد براساس اشتقاق سمت چپ ترینه
گرامر LL(K): وقتی درخت رو داریم بالا به پایین می سازیم و گرامر مربوط به اون درخت است، گرامرهایی که مطرح میشه اینجا گرامرهای LL است - این K میتونه هر عددی از یک یا بیشتر از یک باشه
L اول منظور اینه که از چپ به راست پیمایش میشه اون ورودی (چه درخت بالا به پایین باشه و چه پایین به بالا باشه این ورودی از چپ به راست پیمایش میشه) و L دوم ینی اشتقاق سمت چپ و K هم که به عنوان ورودی میگیره منظور اون تعداد کاراکتر ورودی است که لازمه که ببینه که بتونه تشخیص بده که چه قاعده ای رو انتخاب بکنه
برای k=2 چجوری میشه؟؟؟
نکته: اون چیزی که پارسراها استفاده می کنن همین گرامر LL(1) است چون پیاده سازیشون هم راحت تره

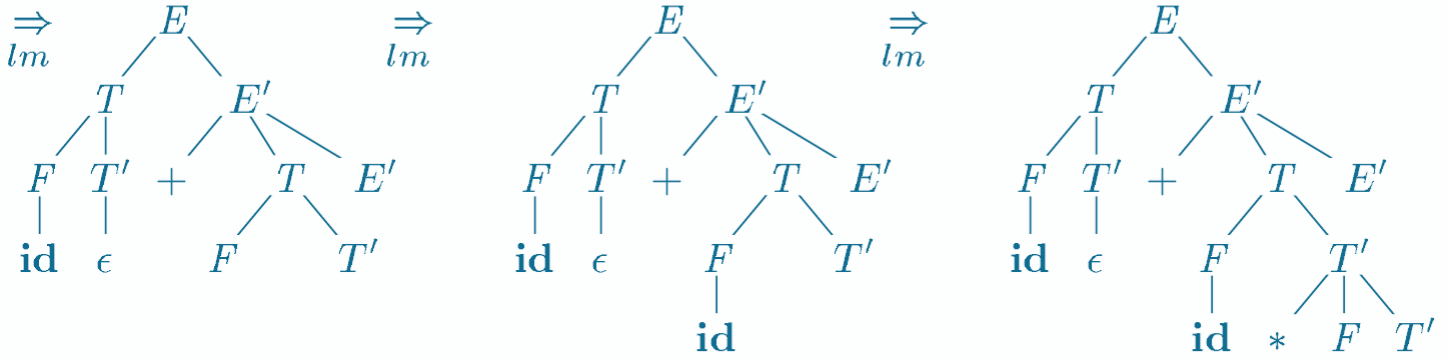
بالا به پایین است

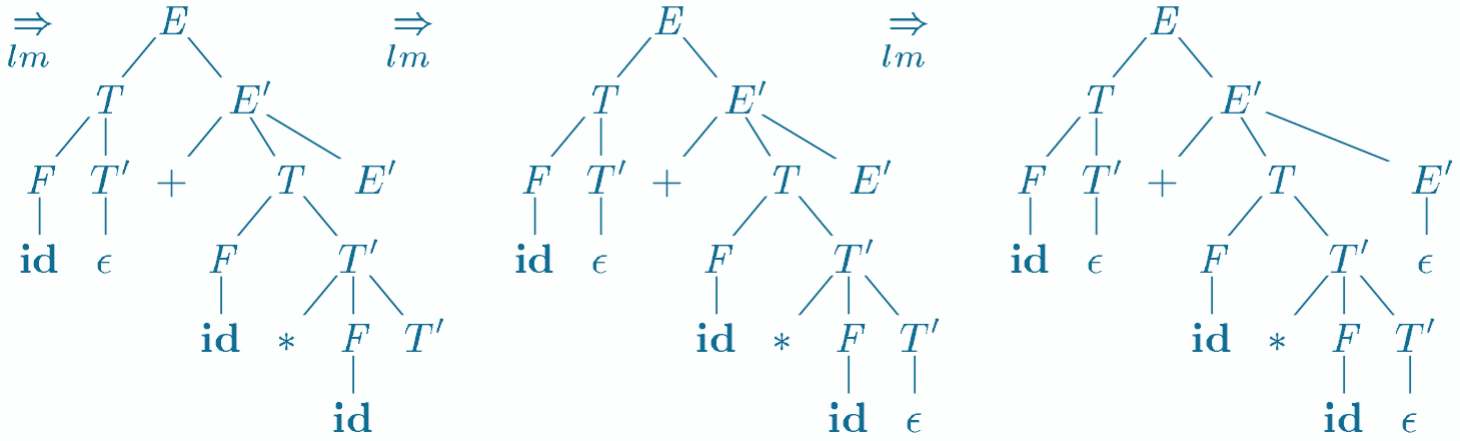


سمت چپ ترین است واسه همین اینو باز کرده و
برای بقیه هم به همین صورت پیش رفته

- **Example**

$$
\begin{array}{lcl}
E & \rightarrow & T\ E' \\
E' & \rightarrow & +\ T\ E' \mid \epsilon \\
T & \rightarrow & F\ T' \\
T' & \rightarrow & *\ F\ T' \mid \epsilon \\
F & \rightarrow & (\ E\ ) \mid \mathbf{id}
\end{array}
$$





تهش به id + id * id رسیده

Recursive-Descent Parsing

- A recursive-descent parsing program consists of a set of procedures, one for each nonterminal
- Execution begins with the procedure for the start symbol

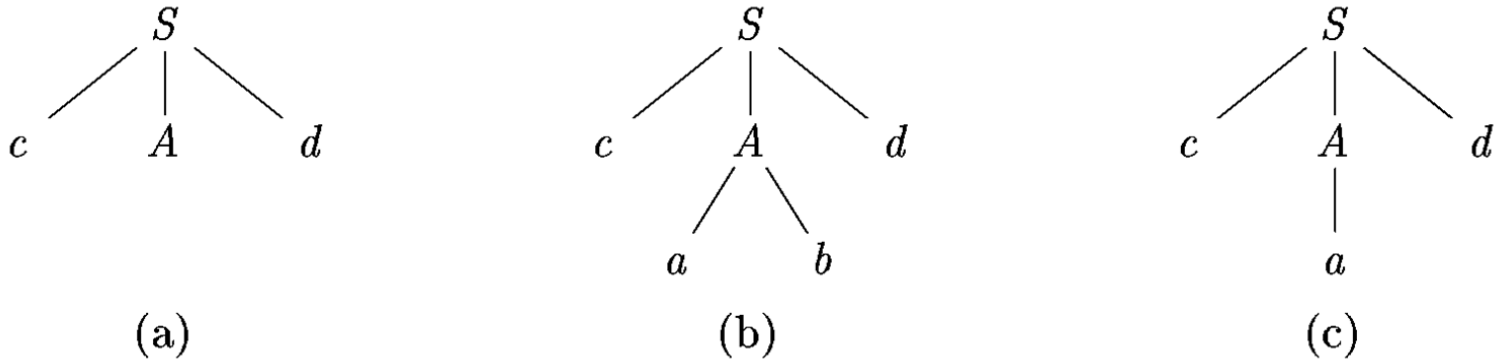
```
     void A() {
1)       Choose an A-production, A → X1 X2 ··· Xk;
2)       for ( i = 1 to k ) {
3)           if ( Xi is a nonterminal )
4)               call procedure Xi();
5)           else if ( Xi equals the current input symbol a )
6)               advance the input to the next symbol;
7)           else /* an error has occurred */;
         }
     }
```

-
Parsing بازگشتی - کاهشی:
برای هر قاعده ای میاد معادل اون رو جایگزین میکنه و برای همه نان ترمینال هایی که سمت راست اون قاعده باشه از چپ به راست شروع میکنه و دوباره جایگزینی انجام میده

Recursive-Descent Parsing

- General recursive-descent may require backtracking; that is, it may require repeated scans over the input
- **Example**
  - Parse tree for $w = cad$

$$S \rightarrow c\ A\ d$$
$$A \rightarrow a\ b \mid a$$



  - In going back to A, we must reset the input pointer to position 2

-
خیلی جاها ممکنه به یک حالتی برسیم که نیاز به backtracking داشته باشیم ینی اون چیزی که بهش رسیدیم درست نبوده و مجبورم برگردیم و دوباره یک قاعده دیگه رو باز کنیم
مثلا توی این گرامر می خواستیم رشته cad رو تولید کنیم ولی رسیدیم به رشته cabd پس اینجا به عقب برمیگردیم که این یک زمانی رو از اون کامپایلر میگیره چون باید برگرده به عقب و دوباره این کاراکتر قبلی رو بخونه ینی A و یک قاعده دیگه رو انتخاب بکنه

FIRST and FOLLOW

- The construction of both top-down and bottom-up parsers is aided by two functions, **FIRST** and **FOLLOW**, associated with a grammar G
- ***FIRST*(*X*) for all grammar symbols *X***

1. If X is a terminal, then $\text{FIRST}(X) = \{X\}$.
2. If X is a nonterminal and $X \to Y_1 Y_2 \cdots Y_k$ is a production for some $k \geq 1$, then place a in $\text{FIRST}(X)$ if for some i, a is in $\text{FIRST}(Y_i)$, and ϵ is in all of $\text{FIRST}(Y_1), \ldots, \text{FIRST}(Y_{i-1})$; that is, $Y_1 \cdots Y_{i-1} \stackrel{*}{\Rightarrow} \epsilon$. If ϵ is in $\text{FIRST}(Y_j)$ for all $j = 1, 2, \ldots, k$, then add ϵ to $\text{FIRST}(X)$. For example, everything in $\text{FIRST}(Y_1)$ is surely in $\text{FIRST}(X)$. If Y_1 does not derive ϵ, then we add nothing more to $\text{FIRST}(X)$, but if $Y_1 \stackrel{*}{\Rightarrow} \epsilon$, then we add $\text{FIRST}(Y_2)$, and so on.
3. If $X \to \epsilon$ is a production, then add ϵ to $\text{FIRST}(X)$.

-
FIRST: با چی می تونه شروع بشه
FOLLOW: چی بعدش می تونه بیاد
حساب کردن FIRST(x):
1- اگر x ترمینال باشه مثلا FIRST(a) ینی رشته a با چی شروع میشه که خیلی واضح است که میشه a
2- اگر x نان ترمینال باشه --> قاعده هایی که توسط اون نان ترمینال تولید میشه رو در نظر میگیریم ؟؟؟
3- اگر قاعده ای داشته باشیم به این صورت که x می دهد اپسیلون، اپسیلون توی FIRST(x) میاد

FIRST and FOLLOW

- ***FOLLOW*(*A*) for all nonterminals *A***

1. Place \$ in FOLLOW(S), where S is the start symbol, and \$ is the input right endmarker.
2. If there is a production $A \to \alpha B \beta$, then everything in FIRST(β) except ϵ is in FOLLOW(B).
3. If there is a production $A \to \alpha B$, or a production $A \to \alpha B \beta$, where FIRST(β) contains ϵ, then everything in FOLLOW(A) is in FOLLOW(B).

++ * +

-
FOLLOW فقط برای نان ترمینال ها حساب میشه
1- اول از همه یک کاراکتری رو که اینجا $ در نظر گرفته میشه به عنوان سمت راست ترین کاراکتر در نظر گرفته میشه --> این کاراکتر که به عنوان کاراکتر انتها در نظر گرفته شده یک کاراکتر خاص است که توی FOLLOW(S) میاد که S هم فرض کردیم متغییر شروع گرامر است پس متغییر شروع هرچی که باشه ما به FOLLOWاش $ رو اضافه میکنیم
2- به ازای هر قاعده ی ++ که داریم همه اون چیزایی که توی FIRST بتا است ینی اون چیزایی که بتا می تونه باهاشون شروع بشه می تونن توی FOLLOW(B) همشون بیان غیر از اپسیلون مثلا FIRST بتا قبلا حساب شده و شده a , b که این دوتا الان توی FOLLOW(B) هم می تونن بیان چون بتا بعد از B کلا اومده توی گرامر --> ینی نگاه میکنیم B کجاها اومده سمت راست قواعد بعد از اون هرچی بود می نویسیم
3- اگر همچین قاعده ای * داشته باشیم ینی بعد از B هیچی نیست یا اگر + رو داشته باشیم ینی بعد از B بتا اومده و فرست این بتا بتونه اپسیلون بشه در این حالت همه اون چیزایی که توی FOLLOW(A) هستن به FOLLOW(B) اضافه میشن

FIRST and FOLLOW

- **Example**

$$
\begin{array}{lcl}
E & \rightarrow & T\ E' \\
E' & \rightarrow & +\ T\ E' \mid \epsilon \\
T & \rightarrow & F\ T' \\
T' & \rightarrow & *\ F\ T' \mid \epsilon \\
F & \rightarrow & (\ E\ ) \mid \mathbf{id}
\end{array}
$$

+

- $FIRST(F) = FIRST(T) = FIRST(E) = \{(, id\}$
- $FIRST(E') = \{+, \epsilon\}$
- $FIRST(T') = \{*, \epsilon\}$
- $FOLLOW(E) = FOLLOW(E') = \{), \$\}$
- $FOLLOW(T) = FOLLOW(T') = \{+,), \$\}$
- $FOLLOW(F) = \{+, *,), \$\}$

برای فرست E میاد فرست سمت چپ قاعده رو میگیره ینی فرست T حالا فرست T میشه فرست F و حالا فرست F میشه id و) که این دوتا ترمینال هستن پس تموم میشه همین جا

برای فالو ابتدا $ به اون گرامری اضافه می کنیم که E داخلش هست چون می خوایم فالو E رو حساب کنیم ینی فالو E داخلش $ هست بعد از اون میایم برای فالو E سمت راست ترینشو در نظر می گیریم ینی اول نگاه میکنیم داخل کدوم گرامر E هست و بعد میایم سمت راست اونو نگاه میکنیم که میشه اینجا + پس فالو E میشه فرست (که در نهایت میشه خود (و خود $ که خودمون اضافه کرده بودیم