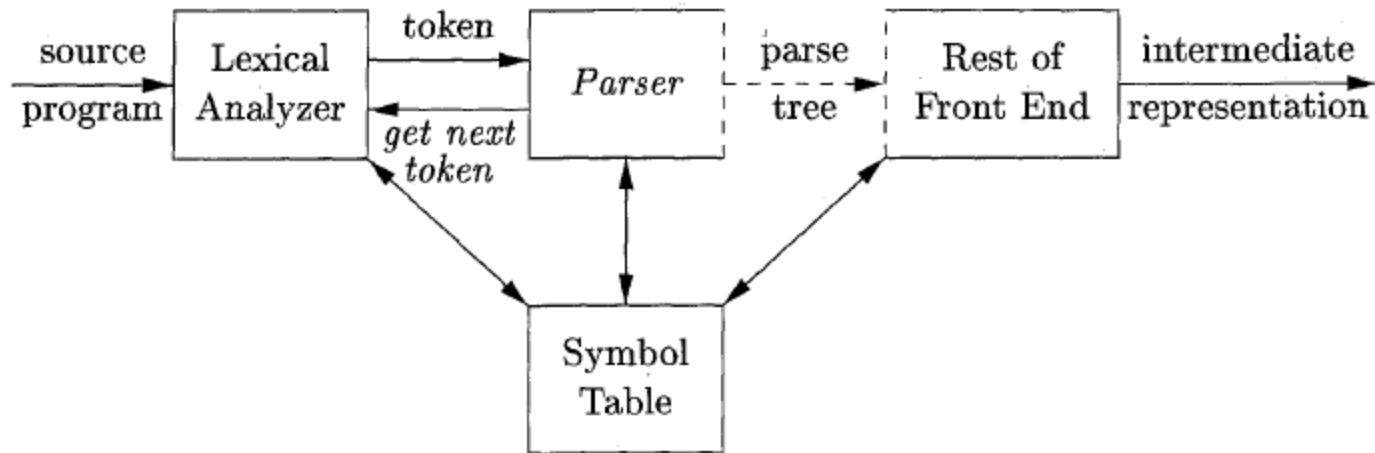


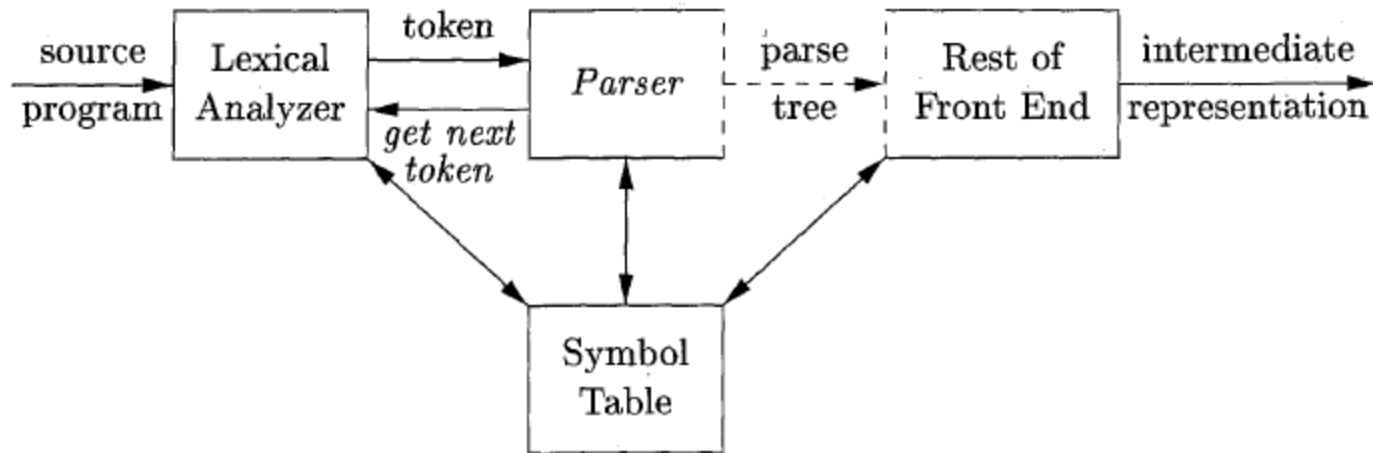
Context-Free Grammars

- Precise syntactic specifications of a programming language
- For some classes, we can construct automatically an efficient parser
- Allows a language to evolve

The Parser



The Parser

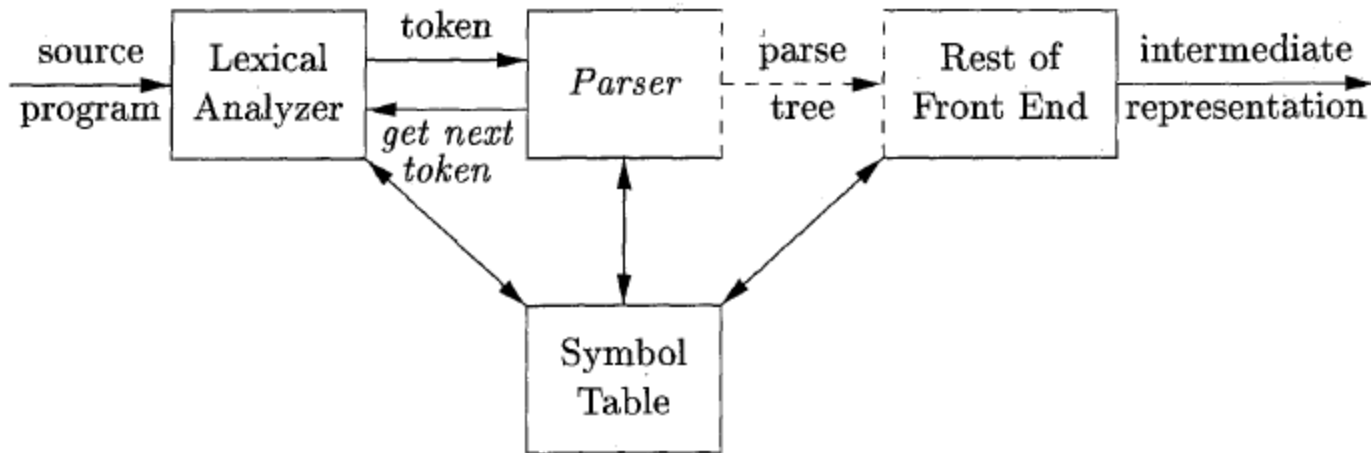


Three general types of parsers

Universal parsing methods:

- can parse any grammar
- too inefficient to use in production compilers

The Parser

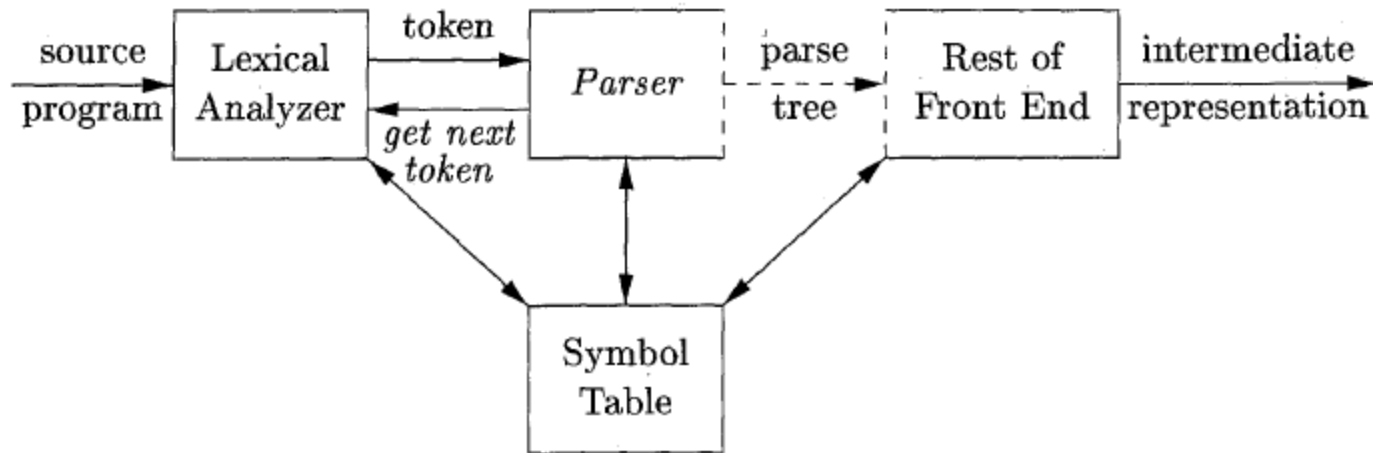


Three general types of parsers

Top-down methods:

- Parse-trees built from root to leaves.
- Input to parser scanned from left to right one symbol at a time

The Parser



Three general types of parsers

Bottom-up methods:

- Start from leaves and work their way up to the root.
- Input to parser scanned from left to right one symbol at a time

Dealing With Errors

If compiler had to process only correct programs, its design and implementation would be simplified greatly!

- Few languages have been designed with error handling in mind.
- Error handling is left to compiler designer.
- Bugs caused about 50% of the total cost, same as they used to be 50 years ago!

Common Programming Errors

- *Lexical errors*: misspellings of identifiers, keywords, or operators
- *Syntactic errors*: misplaced semicolons, extra or missing braces, case without switch,
- *Semantic errors*: type mismatches between operators and operands
- *Logical errors*: anything else!

Wish List

- Report the presence of errors clearly and accurately
- Recover from each error quickly enough to detect subsequent errors
- Add minimal overhead to the processing of correct programs

Easier said than done!

Error-Recovery Strategies

- **Simplest:** quit with an informative error message when detecting the first error
- **Panic-mode Recovery:** discards input symbols one at a time until a designated synchronizing tokens is found.
- **Phrase-level Recovery:** perform local correction on the remaining input. The choice of local correction is left to the compiler designer.
- **Error Production:** production rules for common errors.

Context-Free Grammar

Terminals
(token name)

Nonterminals

Example:

expression → *expression* + *term*
expression → *expression* - *term*
expression → *term*
 term → *term* * *factor*
 term → *term* / *factor*
 term → *factor*
factor → (*expression*)
factor → **id**

Start
Symbol

Productions

Derivations

- Starting with start symbol
- At each step: a nonterminal replaced with the body of a production

Example:

$$E \rightarrow E + E \mid E * E \mid - E \mid (E) \mid \text{id}$$

Deriving: **$-(\text{id} + \text{id})$**

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E + E) \Rightarrow -(\text{id} + E) \Rightarrow -(\text{id} + \text{id})$$

More on Derivations

\Rightarrow means derive in one step

\Rightarrow^* means derive in zero or more steps

\Rightarrow^+ means derive in one or more steps

1. $\alpha \Rightarrow^* \alpha$, for any string α , and
2. If $\alpha \Rightarrow^* \beta$ and $\beta \Rightarrow \gamma$, then $\alpha \Rightarrow^* \gamma$.

Leftmost derivations, the leftmost nonterminal in each sentential is always chosen. $\alpha \underset{lm}{\Rightarrow} \beta$

Rightmost derivations, the rightmost nonterminal in each sentential is always chosen. $\alpha \underset{rm}{\Rightarrow} \beta$

Example

For the context-free grammar:

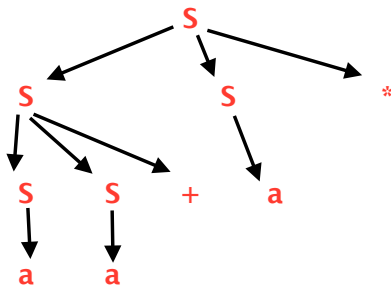
$$S \rightarrow S S + \mid S S * \mid a$$

and the string $aa + a*$.

- a) Give a leftmost derivation for the string.
- b) Give a rightmost derivation for the string.
- c) Give a parse tree for the string.

$S = lm \Rightarrow SS^* \Rightarrow SS+S^* \Rightarrow aS+S^* \Rightarrow aa+S^* \Rightarrow aa+a^*$

$S = rm \Rightarrow SS^* \Rightarrow Sa^* \Rightarrow SS+a^* \Rightarrow Sa+a^* \Rightarrow aa+a^*$



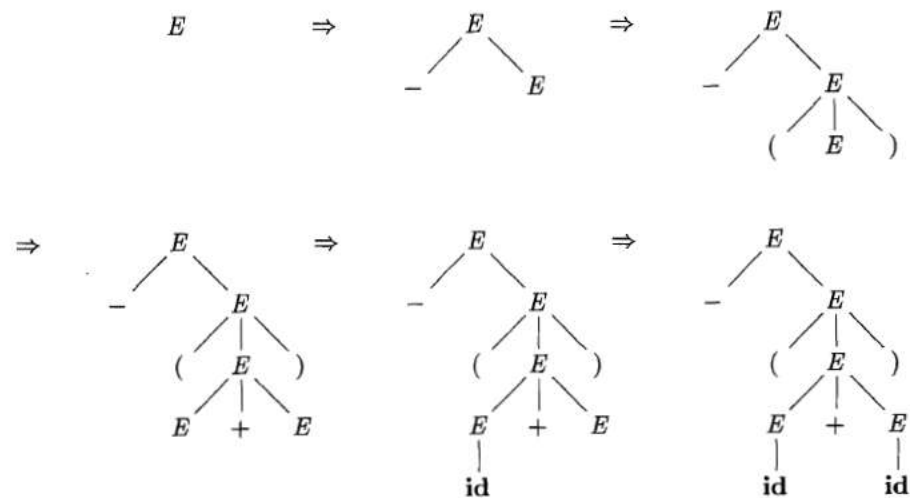
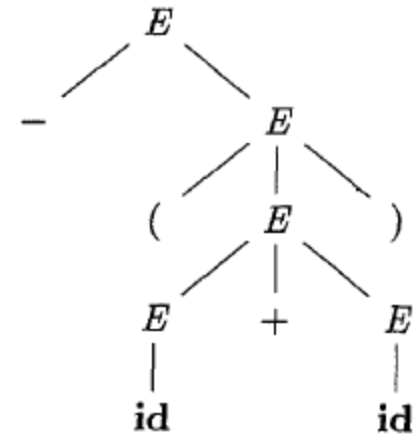
Parse Trees

- What is the relationship between a parse-tree and derivations?
 - Parse tree is the graphical representation of derivations
 - Filters out order of nonterminal replacement
 - many-to-one relationship between derivations and parse-tree

$$E \rightarrow E + E \mid E * E \mid - E \mid (E) \mid \text{id}$$

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E + E) \Rightarrow -(E + \text{id}) \Rightarrow -(\text{id} + \text{id})$$

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E + E) \Rightarrow -(\text{id} + E) \Rightarrow -(\text{id} + \text{id})$$



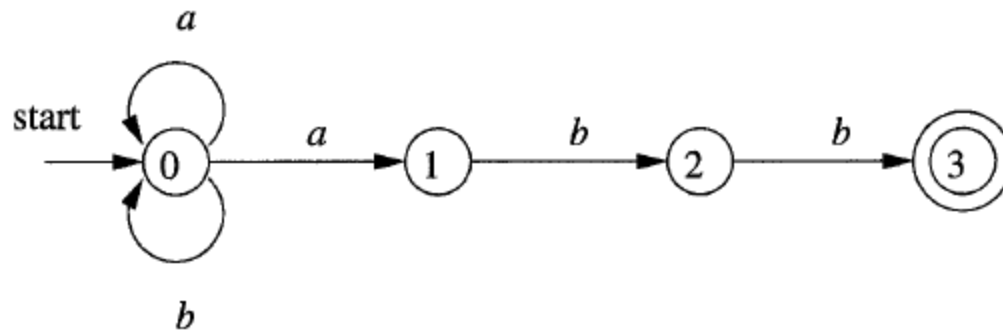
Context-Free Grammar Vs Regular Expressions

- Grammars are more powerful notations than regular expressions
 - Every construct that can be described by a regular expression can be described by a grammar, but not vice-versa

Regular expression \rightarrow NFA then:

1. For each state i of the NFA, create a nonterminal A_i .
2. If state i has a transition to state j on input a , add the production $A_i \rightarrow aA_j$. If state i goes to state j on input ϵ , add the production $A_i \rightarrow A_j$.
3. If i is an accepting state, add $A_i \rightarrow \epsilon$.
4. If i is the start state, make A_i be the start symbol of the grammar.

$(a | b)^*abb$



$$\begin{aligned} A_0 &\rightarrow aA_0 \mid bA_0 \mid aA_1 \\ A_1 &\rightarrow bA_2 \\ A_2 &\rightarrow bA_3 \\ A_3 &\rightarrow \epsilon \end{aligned}$$

Question Worth Asking

If grammars are much more powerful than regular expressions, why not using them in lexical analysis too?

- Lexical rules are quite simple and do not need notation as powerful as grammars
- Regular expressions are more concise and easier to understand for tokens
- More efficient lexical analyzers can be generated from regular expressions than from grammars

How Can We Enhance Our Grammar?

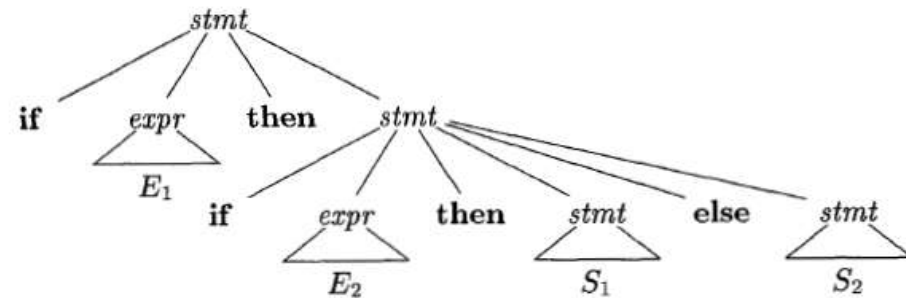
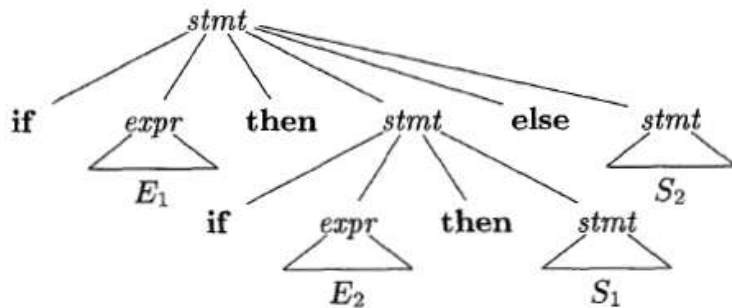
- Eliminating ambiguity
- Eliminating left-recursion
- Left factoring

Eliminating Ambiguity

Sometimes we can re-write grammar to eliminate ambiguity

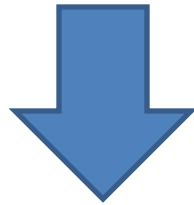
$stmt \rightarrow$ if $expr$ then $stmt$
|
if $expr$ then $stmt$ else $stmt$
|
other

if E_1 then if E_2 then S_1 else S_2



if E_1 then if E_2 then S_1 else S_2

stmt → **if *expr* then *stmt***
| **if *expr* then *stmt* else *stmt***
| **other**



stmt → *matched_stmt*
| *open_stmt*
matched_stmt → **if *expr* then *matched_stmt* else *matched_stmt***
| **other**
open_stmt → **if *expr* then *stmt***
| **if *expr* then *matched_stmt* else *open_stmt***

Eliminating Left-Recursion

$$A \rightarrow A\alpha \mid \beta \quad \longrightarrow \quad \begin{array}{l} A \rightarrow \beta A' \\ A' \rightarrow \alpha A' \mid \epsilon \end{array}$$

$$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \dots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \dots \mid \beta_n$$

$$\begin{array}{l} A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \dots \mid \beta_n A' \\ A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \dots \mid \alpha_m A' \mid \epsilon \end{array}$$

How about something like:

$$\begin{array}{l} S \rightarrow A a \mid b \\ A \rightarrow A c \mid S d \mid \epsilon \end{array}$$

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

$$\begin{array}{l} S \rightarrow A a \mid b \\ A \rightarrow b d A' \mid A' \\ A' \rightarrow c A' \mid a d A' \mid \epsilon \end{array}$$

$E \rightarrow E+T \mid T$

we can assume

$A=E$

$+T= \text{alpha}$

$T= B$

Left-Factoring

- A way of delaying the decision until more info is available

$$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \quad \longrightarrow \quad \begin{array}{l} A \rightarrow \alpha A' \\ A' \rightarrow \beta_1 \mid \beta_2 \end{array}$$

Example: $stmt \rightarrow \begin{array}{l} \text{if } expr \text{ then } stmt \text{ else } stmt \\ \text{if } expr \text{ then } stmt \end{array}$

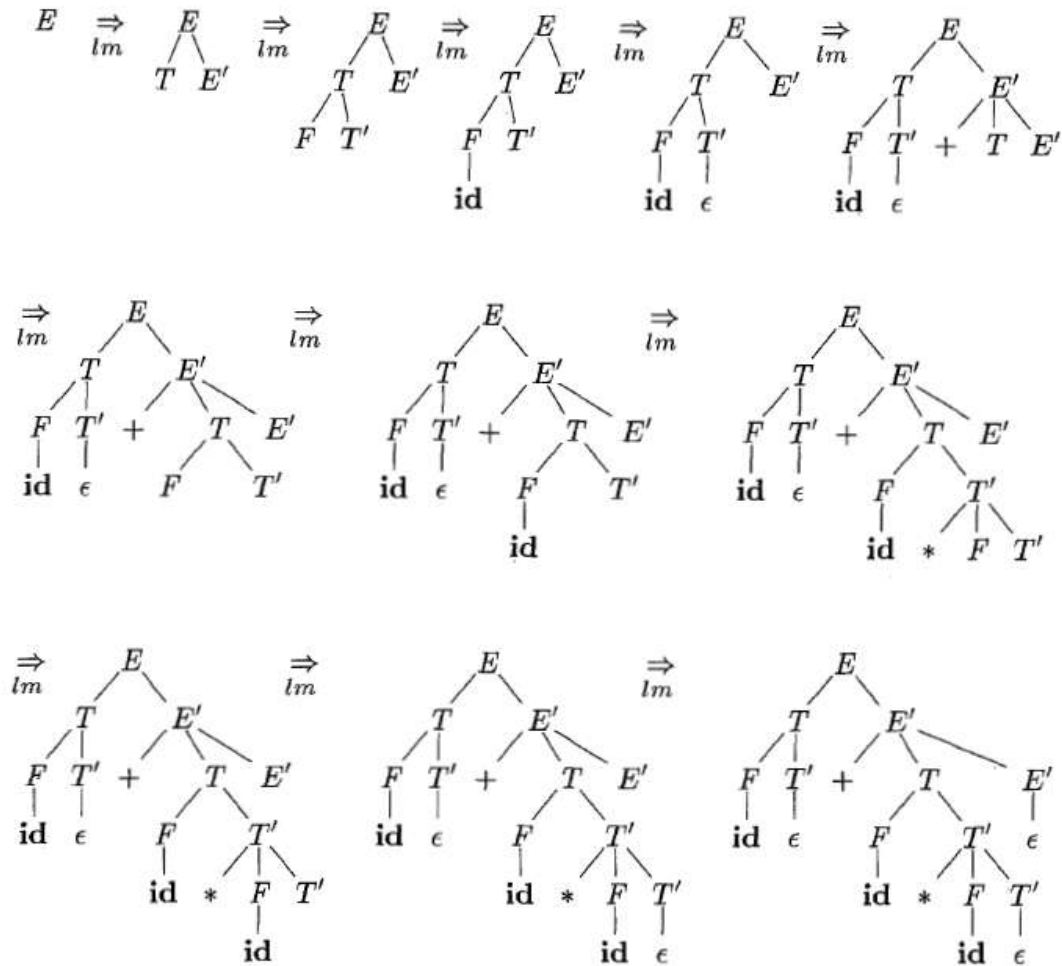


$stmt \rightarrow EXP \text{ else } stmt \mid EXP$
 $EXP \rightarrow \text{if } expr \text{ then } stmt$

Top-Down Parsing

- Constructing a parse tree for an input string starting from root
 - Parse tree built in preorder (depth-first)
- Finding left-most derivation
- At each step of a top-down parse:
 - determine the production to be applied
 - matching terminal symbols in production body with input string

Given: $E \rightarrow T E'$ and: **id+id*id**
 $E' \rightarrow + T E' \mid \epsilon$
 $T \rightarrow F T'$
 $T' \rightarrow * F T' \mid \epsilon$
 $F \rightarrow (E) \mid \text{id}$




Recursive - Descent Parsing

recursive descent parser is a kind of top-down parser built from a set of mutually recursive procedures (or a non-recursive equivalent) where each such procedure usually implements one of the productions of the grammar.

Recursive-Descent Parsing

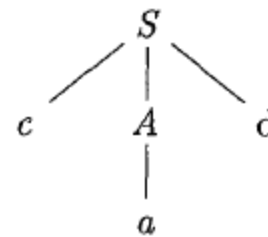
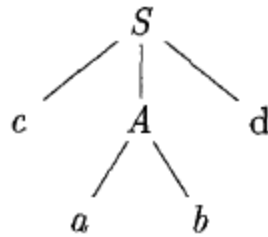
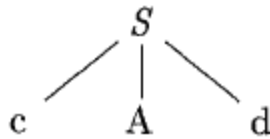
How?



```
void A() {  
1)    Choose an  $A$ -production,  $A \rightarrow X_1X_2 \cdots X_k$ ;  
2)    for (  $i = 1$  to  $k$  ) {  
3)        if (  $X_i$  is a nonterminal )  
4)            call procedure  $X_i()$ ;  
5)        else if (  $X_i$  equals the current input symbol  $a$  )  
6)            advance the input to the next symbol;  
7)        else /* an error has occurred */;  
    }  
}
```

Example of Backtracking

$S \rightarrow c A d$ and input $w = cad$,
 $A \rightarrow a b \mid a$



Important Concepts: FIRST and FOLLOW

Define $FIRST(\alpha)$, where α is any string of grammar symbols, to be the set of terminals that begin strings derived from α . If $\alpha \xRightarrow{*} \epsilon$, then ϵ is also in $FIRST(\alpha)$.

Define $FOLLOW(A)$, for nonterminal A , to be the set of terminals a that can appear immediately to the right of A in some sentential form; that is, the set of terminals a such that there exists a derivation of the form $S \xRightarrow{*} \alpha A a \beta$,

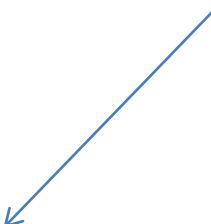
Example

		FIRST	FOLLOW
E	$\rightarrow T E'$	(id)\$
E'	$\rightarrow + T E' \mid \epsilon$	+ ϵ)\$
T	$\rightarrow F T'$	(id	+
T'	$\rightarrow * F T' \mid \epsilon$	* ϵ	+
F	$\rightarrow (E) \mid \mathbf{id}$	(id	* +

LL(1) Grammars

- For recursive-descent parsers with no backtracking
- L = scan from left to right
- L = left-most derivation
- 1 symbol lookahead
- Cannot be left-recursive or ambiguous
- If $A \rightarrow F \mid T$
 - $FIRST(F)$ and $FIRST(T)$ are disjoint
 - if ϵ is in $FIRST(T)$ then $FIRST(F)$ and $FOLLOW(A)$ are disjoint ... and likewise when ϵ is in $FIRST(F)$

Parsing Table



```
void A() {  
1)   Choose an A-production,  $A \rightarrow X_1X_2 \cdots X_k$ ;  
2)   for (  $i = 1$  to  $k$  ) {  
3)       if (  $X_i$  is a nonterminal )  
4)           call procedure  $X_i()$ ;  
5)       else if (  $X_i$  equals the current input symbol  $a$  )  
6)           advance the input to the next symbol;  
7)       else /* an error has occurred */;  
   }  
}
```

Parsing Table

- Two dimensional array
 - Rows: nonterminals Columns: input symbols
- $M[A,a]$ where A is nonterminal and a is terminal or $\$$
- Gives the production rule to use.

1. For each terminal a in $\text{FIRST}(A)$, add $A \rightarrow \alpha$ to $M[A, a]$.
2. If ϵ is in $\text{FIRST}(\alpha)$, then for each terminal b in $\text{FOLLOW}(A)$, add $A \rightarrow \alpha$ to $M[A, b]$. If ϵ is in $\text{FIRST}(\alpha)$ and $\$$ is in $\text{FOLLOW}(A)$, add $A \rightarrow \alpha$ to $M[A, \$]$ as well.

First	Follow			
(id)\$	E	\rightarrow	$T E'$
+ ϵ)\$	E'	\rightarrow	$+ T E' \mid \epsilon$
(id	+)\$	T	\rightarrow	$F T'$
* ϵ	+)\$	T'	\rightarrow	$* F T' \mid \epsilon$
(id	* +)\$	F	\rightarrow	$(E) \mid \text{id}$

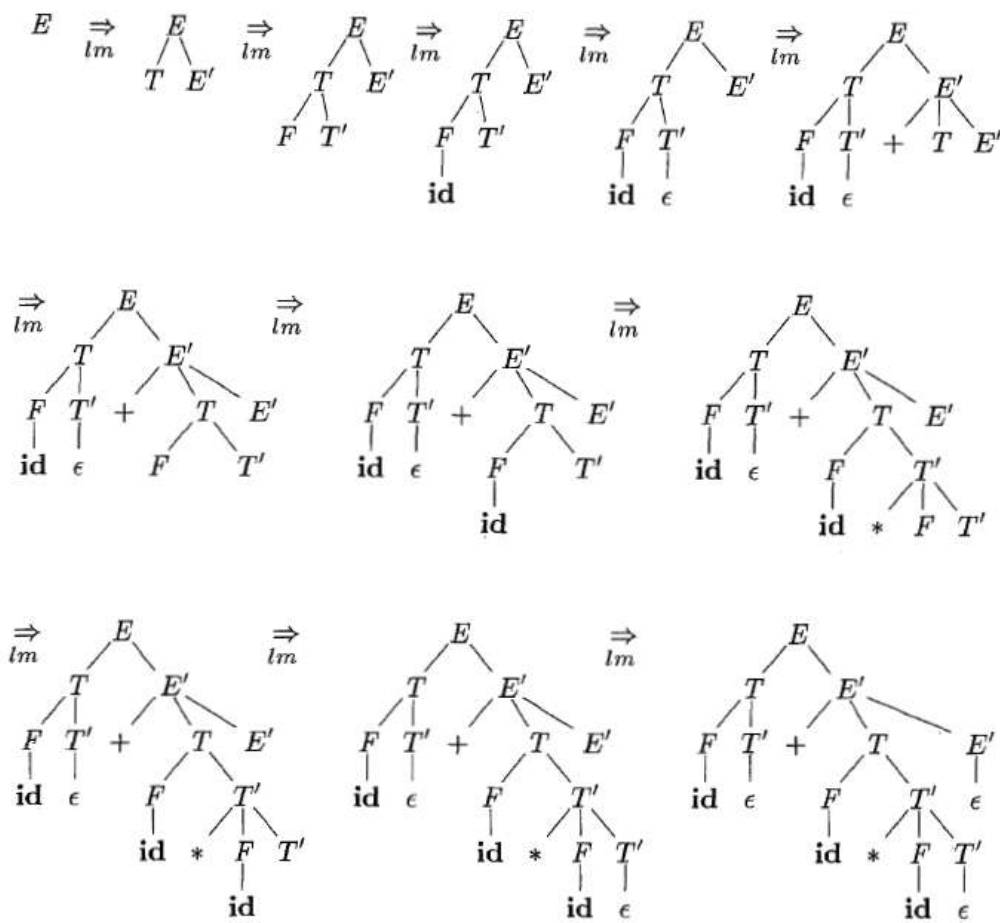
1. For each terminal a in $\text{FIRST}(A)$, add $A \rightarrow \alpha$ to $M[A, a]$.
2. If ϵ is in $\text{FIRST}(\alpha)$, then for each terminal b in $\text{FOLLOW}(A)$, add $A \rightarrow \alpha$ to $M[A, b]$. If ϵ is in $\text{FIRST}(\alpha)$ and $\$$ is in $\text{FOLLOW}(A)$, add $A \rightarrow \alpha$ to $M[A, \$]$ as well.



NON - TERMINAL	INPUT SYMBOL					
	id	+	*	()	\$
E	$E \rightarrow T E'$			$E \rightarrow T E'$		
E'		$E' \rightarrow + T E'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow F T'$			$T \rightarrow F T'$		
T'		$T' \rightarrow \epsilon$	$T' \rightarrow * F T'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow \text{id}$			$F \rightarrow (E)$		

NON - TERMINAL	INPUT SYMBOL					
	id	+	*	()	\$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow \text{id}$			$F \rightarrow (E)$		

id+id*id



Exercise

For the following productions:

$$S \rightarrow +SS \mid *SS \mid a$$

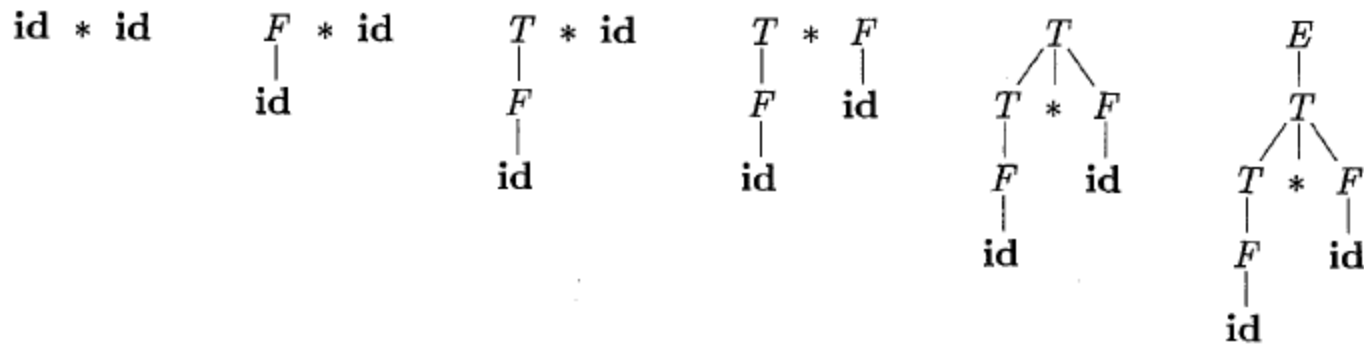
- Write predictive parser
- Write parsing table
- Show how to parse: $+^*aaa$

Bottom-Up Parsing

- Given a string of terminals
- Build parse tree starting from leaves and working up toward the root
- reverse of right-most derivation
- Used for type of grammars called LR
- LR parsers are difficult to build by hand
- We use automatic parser generators for LR grammars

Given: $E \rightarrow E + T \mid T$ and the string: **id * id**
 $T \rightarrow T * F \mid F$
 $F \rightarrow (E) \mid \text{id}$

$E \Rightarrow T \Rightarrow T * F \Rightarrow T * \text{id} \Rightarrow F * \text{id} \Rightarrow \text{id} * \text{id}$



RIGHT SENTENTIAL FORM	HANDLE	REDUCING PRODUCTION
$\text{id}_1 * \text{id}_2$	id_1	$F \rightarrow \text{id}$
$F * \text{id}_2$	F	$T \rightarrow F$
$T * \text{id}_2$	id_2	$F \rightarrow \text{id}$
$T * F$	$T * F$	$E \rightarrow T * F$

Shift-Reduce Parsing

- Form of bottom-up parsing
- Consists of:
 - Stack: holds grammar symbols
 - input buffer: holds the rest of the string to be parsed
- Handle always appears on the top of the stack

Initial position:

STACK
\$

INPUT
 w \$

Final position (success)

STACK
\$ S

INPUT
\$

Actions: **shift, reduce, accept, error**

$$\begin{aligned}
 E &\rightarrow E + T \mid T \\
 T &\rightarrow T * F \mid F \\
 F &\rightarrow (E) \mid \mathbf{id}
 \end{aligned}$$

id * id

STACK	INPUT	ACTION
\$	id₁ * id₂ \$	shift
\$id₁	* id₂ \$	reduce by $F \rightarrow \mathbf{id}$
\$F	* id₂ \$	reduce by $T \rightarrow F$
\$T	* id₂ \$	shift
\$T *	id₂ \$	shift
\$T * id₂	\$	reduce by $F \rightarrow \mathbf{id}$
\$T * F	\$	reduce by $T \rightarrow T * F$
\$T	\$	reduce by $E \rightarrow T$
\$E	\$	accept

Exercise

Let's apply shift-reduce to the following

input: 00S11

and the following productions:

$S \rightarrow 0S1 \mid 01$

So...

- Skim: 4.2.6, 4.3.5, 4.4.4, 4.4.5
- Read rest of 4.1 to 4.5