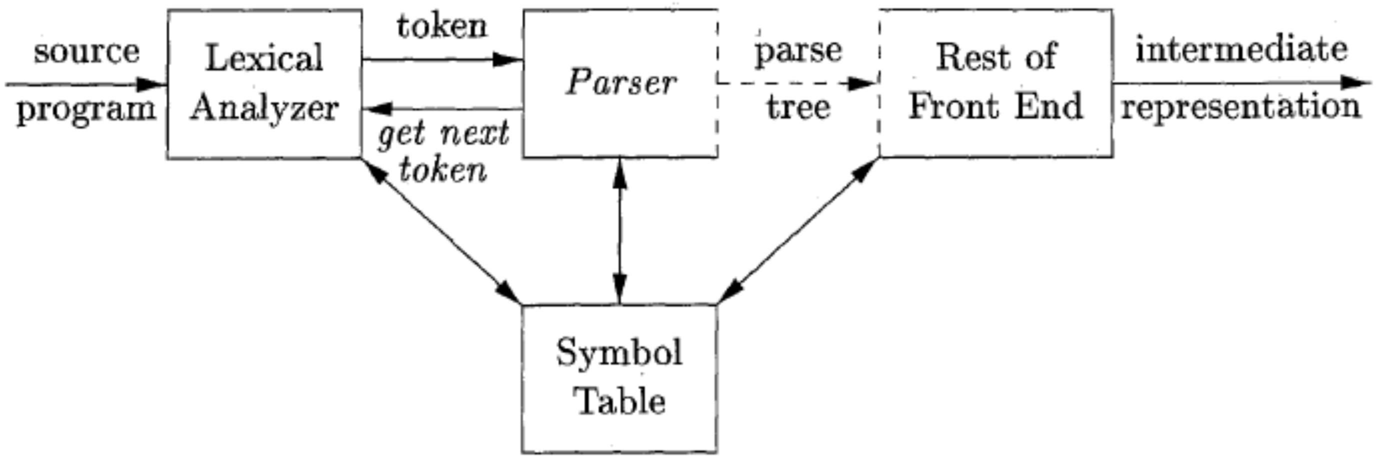
Context-Free Grammars

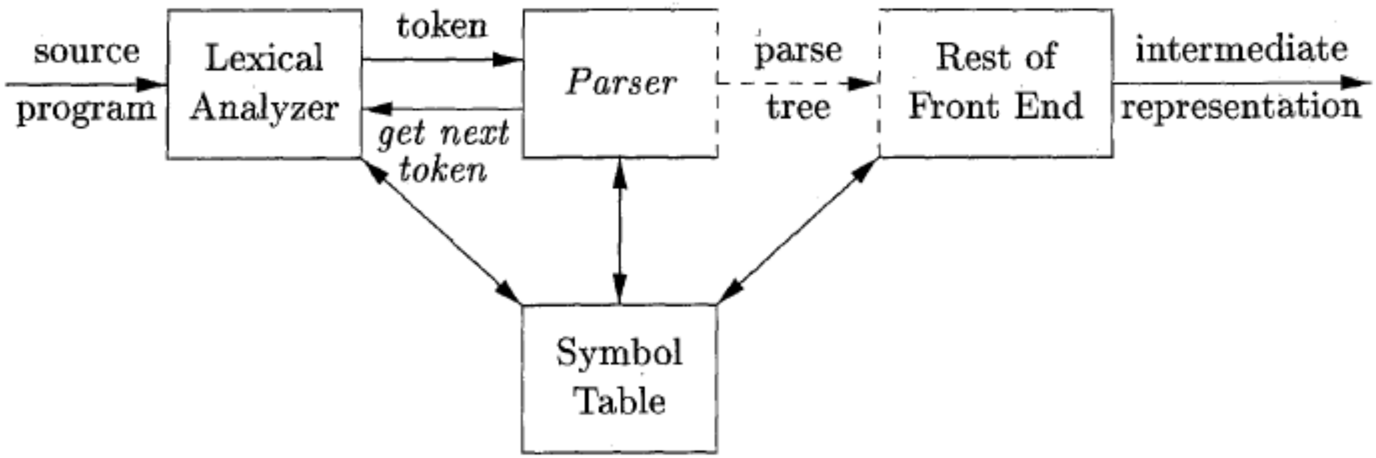
- Precise syntactic specifications of a programming language
- For some classes, we can construct automatically an efficient parser
- Allows a language to evolve

The Parser



source
program
Lexical
Analyzer
token
get next
token
Parser
parse
tree
Rest of
Front End
intermediate
representation
Symbol
Table

The Parser

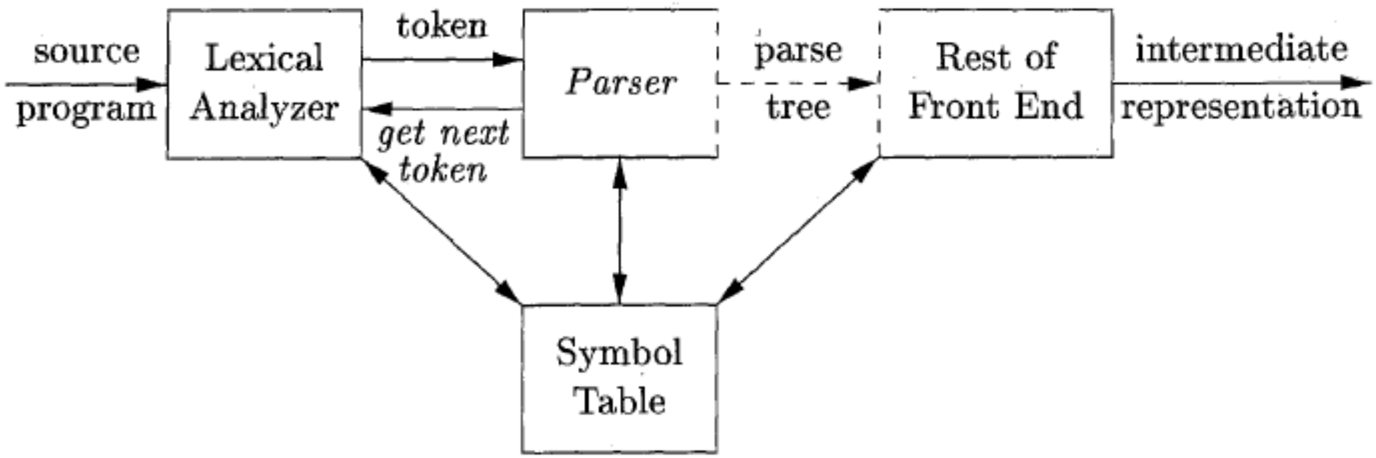


Three general types of parsers

Universal parsing methods:

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

The Parser

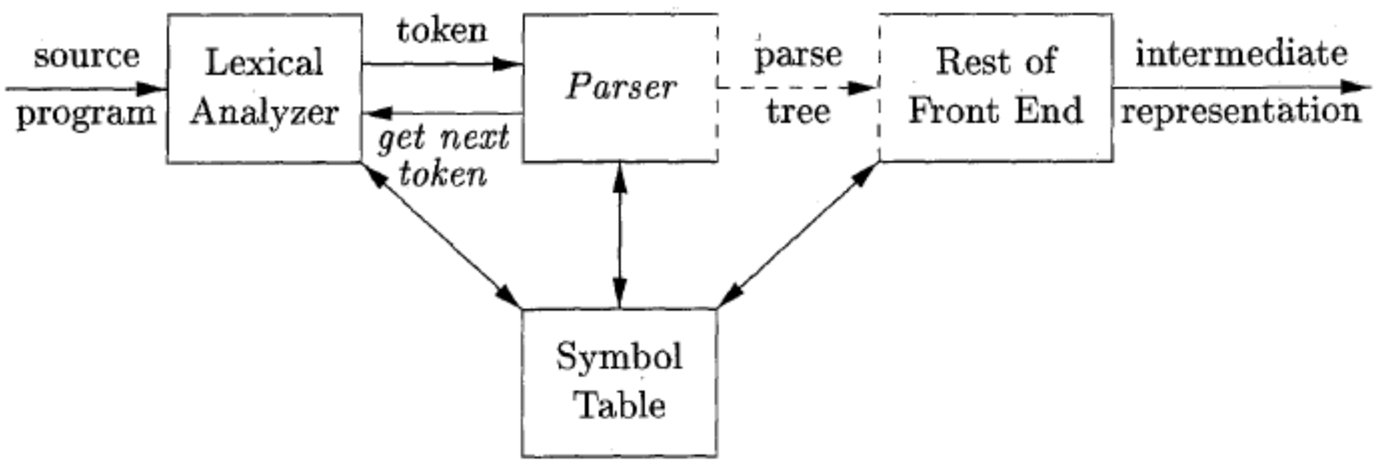


Three general types of parsers

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:

$$
\begin{array}{rcl}
expression & \rightarrow & expression + term \\
expression & \rightarrow & expression - term \\
expression & \rightarrow & term \\
term & \rightarrow & term * factor \\
term & \rightarrow & term \, / \, factor \\
term & \rightarrow & factor \\
factor & \rightarrow & (\ expression\ ) \\
factor & \rightarrow & \mathbf{id}
\end{array}
$$

Start
Symbol

Productions

Derivations

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

Example:

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

Deriving: ***-(id + id)***

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

More on Derivations

$\Rightarrow$ means derive in one step

$\stackrel{*}{\Rightarrow}$ means derive in zero or more steps

$\stackrel{+}{\Rightarrow}$ means derive in one or more steps

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

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 $aa + a*$.

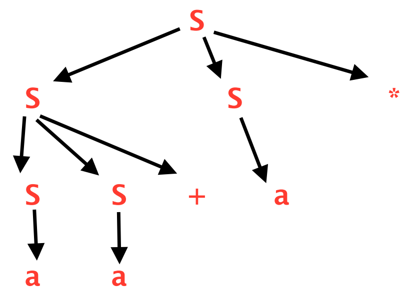
a) Give a leftmost derivation for the string.

b) Give a rightmost derivation for the string.

c) Give a parse tree for the string.

S =lm=> SS* => SS+S* => aS+S* => aa+S* => aa+a*

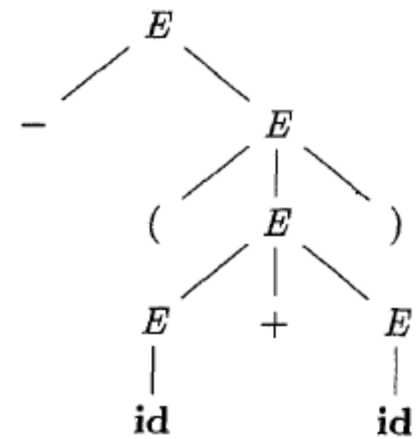
S =rm=> SS* => Sa* => SS+a* => Sa+a* => aa+a*



Parse Trees

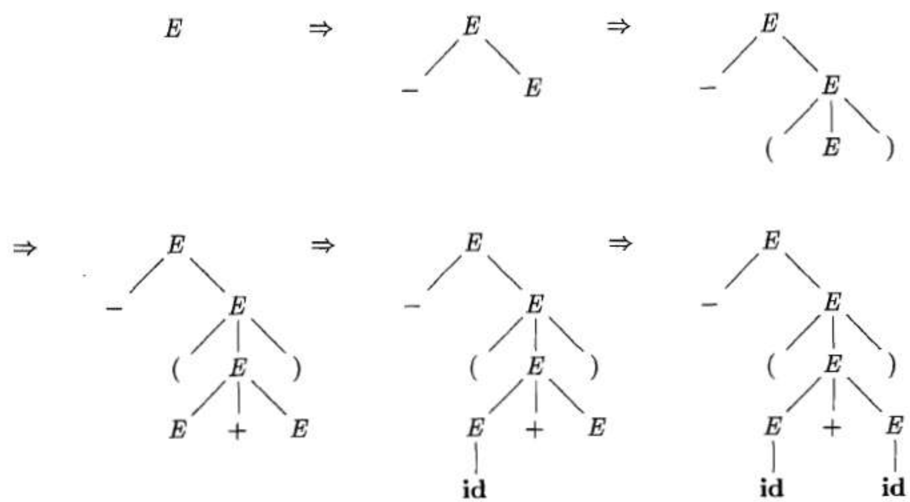
- What is the relationship between a parse-tree and derivations?
  - Parse tree is the graphical representation of derivations
  - Filters out order of nonterminal replacement
  - many-to-one relationship between derivations and parse-tree

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



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

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



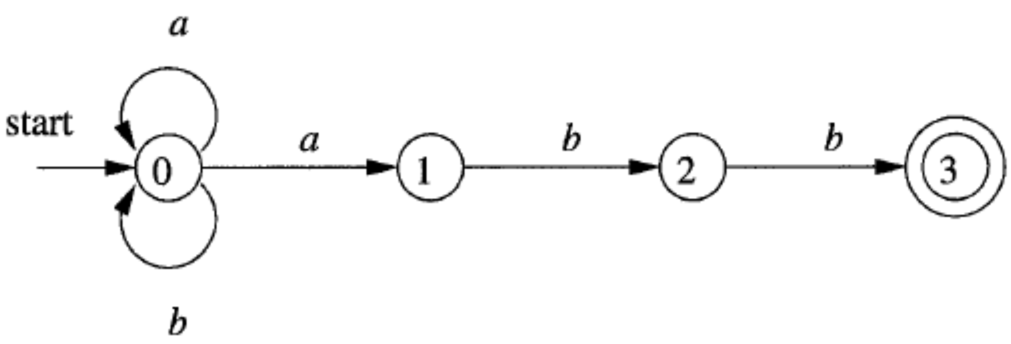
Context-Free Grammar Vs Regular Expressions

- 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 -> 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 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?

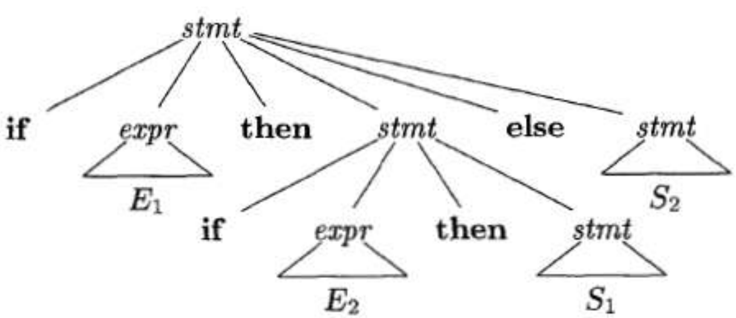
- Eliminating ambiguity
- Eliminating left-recursion
- Left factoring

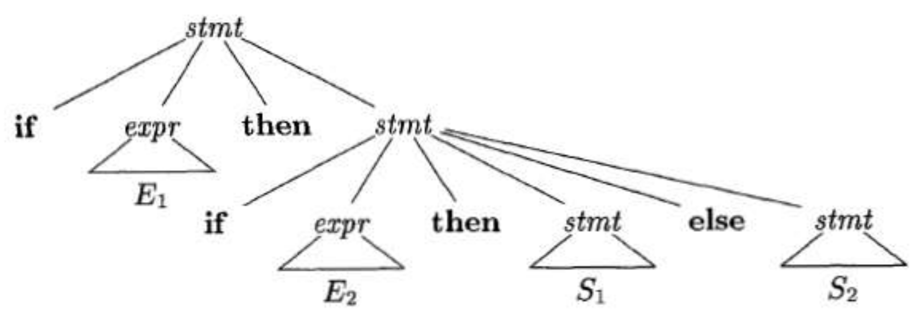
Eliminating Ambiguity

Sometimes we can re-write grammar to eliminate ambiguity

$$\begin{array}{lcl} stmt & \to & \textbf{if } expr \textbf{ then } stmt \\ & | & \textbf{if } expr \textbf{ then } stmt \textbf{ else } stmt \\ & | & \textbf{other} \end{array}$$

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

$$
\begin{array}{lcl}
stmt & \rightarrow & \textbf{if } expr \textbf{ then } stmt \\
 & | & \textbf{if } expr \textbf{ then } stmt \textbf{ else } stmt \\
 & | & \textbf{other}
\end{array}
$$

$$
\begin{array}{rcl}
stmt & \rightarrow & matched\_stmt \\
 & | & open\_stmt \\
matched\_stmt & \rightarrow & \textbf{if } expr \textbf{ then } matched\_stmt \textbf{ else } matched\_stmt \\
 & | & \textbf{other} \\
open\_stmt & \rightarrow & \textbf{if } expr \textbf{ then } stmt \\
 & | & \textbf{if } expr \textbf{ then } matched\_stmt \textbf{ else } open\_stmt
\end{array}
$$

Eliminating Left-Recursion

$$A \to A\alpha \mid \beta \quad \Rightarrow \quad \begin{array}{l} A \to \beta A' \\ A' \to \alpha A' \mid \epsilon \end{array}$$

$$A \to A\alpha_1 \mid A\alpha_2 \mid \cdots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n$$

$$\Downarrow$$

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

How about something like:

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

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

E -> E+T |T

we can assume

A=E

+T= alpha

T= B

Left-Factoring

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

$$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \quad \Rightarrow \quad \begin{aligned} A &\rightarrow \alpha A' \\ A' &\rightarrow \beta_1 \mid \beta_2 \end{aligned}$$

Example:

$$\begin{aligned} \mathit{stmt} \rightarrow & \ \textbf{if}\ \mathit{expr}\ \textbf{then}\ \mathit{stmt}\ \textbf{else}\ \mathit{stmt} \\ \mid & \ \textbf{if}\ \mathit{expr}\ \textbf{then}\ \mathit{stmt} \end{aligned}$$

⇓

stmt -> EXP ***else*** *stmt | EXP*
*EXP -> **if** expr **then** stmt*

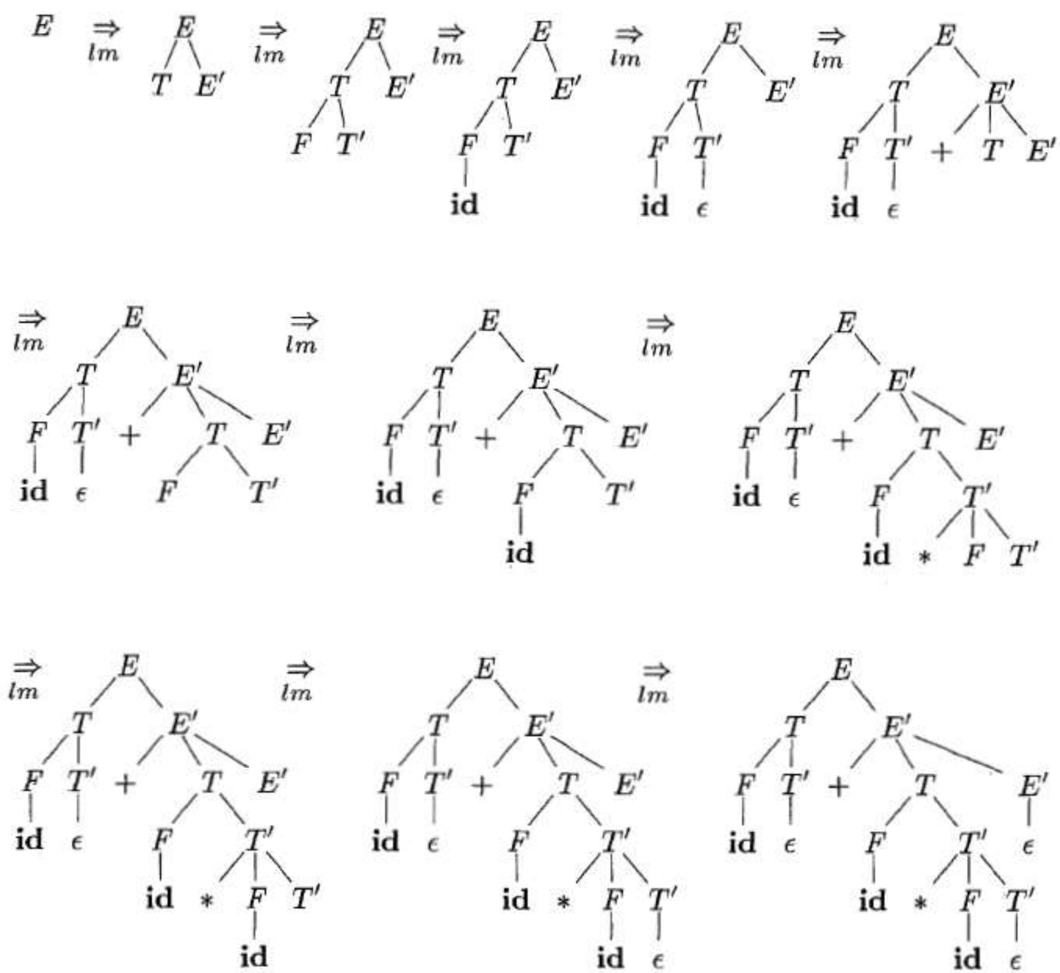
Top-Down Parsing

- 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:

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

and: **id+id*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

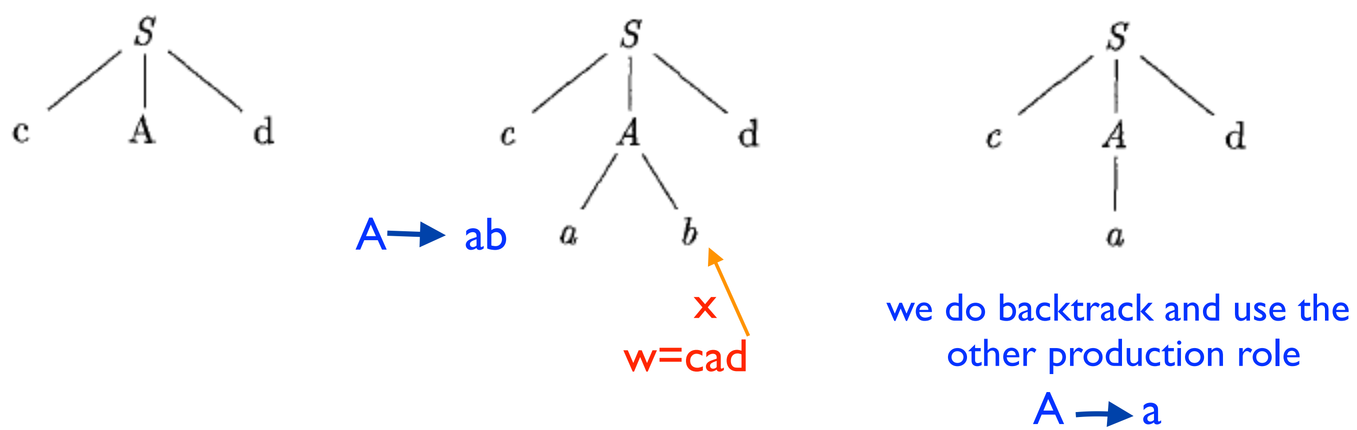
How?

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

Example of Backtracking

$$
\begin{array}{lll}
S & \rightarrow & c\ A\ d \\
A & \rightarrow & a\ b\ \mid\ a
\end{array}
$$

and input $w = cad$,



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 $\alpha \stackrel{*}{\Rightarrow} \epsilon$, then ϵ is also in FIRST(α).

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 $S \stackrel{*}{\Rightarrow} \alpha A a \beta$,

Example

Production	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	* +) $

for the followT: we have E' , what is the first thing we hit in E'?
it is + | ε ,
NOTE: we cannot have ε in the Follow.
we added the + to the follow .
ε can substitute E' in role #1.
the new follow will be the follow of E , which is)$

F'
T' =>* | ε

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-> F | 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() {
    Choose an A-production, A → X1 X2 ··· Xk;
    for ( i = 1 to k ) {
        if ( Xi is a nonterminal )
            call procedure Xi();
        else if ( Xi equals the current input symbol a )
            advance the input to the next symbol;
        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 FIRST(A), add $A \rightarrow \alpha$ to $M[A, a]$.
2. If ϵ is in FIRST(α), then for each terminal b in FOLLOW(A), add $A \rightarrow \alpha$ to $M[A, b]$. If ϵ is in FIRST(α) and \$ is in FOLLOW(A), add $A \rightarrow \alpha$ to $M[A, \$]$ as well.

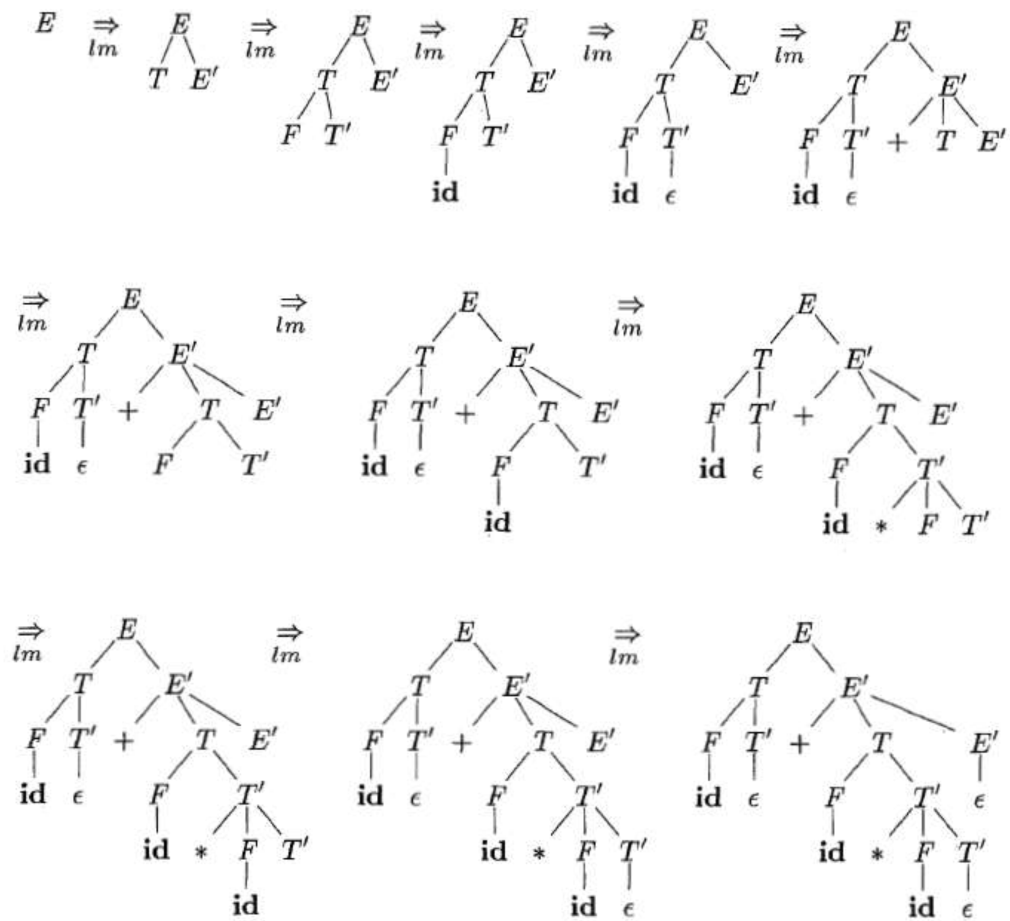
First	Follow	Production
(id	)$	$E \rightarrow T\ E'$
+ ε	)$	$E' \rightarrow +\ T\ E' \mid \epsilon$
(id	+) $	$T \rightarrow F\ T'$
* ε	+) $	$T' \rightarrow *\ F\ T' \mid \epsilon$
(id	* +) $	$F \rightarrow (\ E\ ) \mid \mathbf{id}$

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

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

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

id+id*id



Exercise

For the following productions:

S-> +SS | * SS | a

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

Bottom-Up Parsing

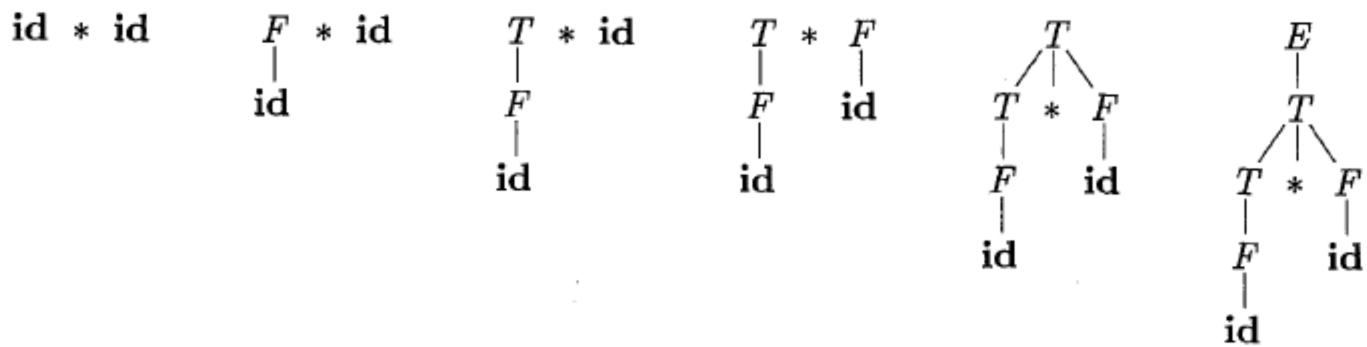
- Given a string of terminals
- Build parse tree starting from leaves and working up toward the root
- reverse of right-most derivation
- Used for type of grammars called LR
- LR parsers are difficult to build by hand
- We use automatic parser generators for LR grammars

Given:

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

and the string: $\mathbf{id} * \mathbf{id}$

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



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

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	INPUT
$\$\,S$	$\$$

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

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

$\mathbf{id} * \mathbf{id}$

STACK	INPUT	ACTION
$\$$	$\mathbf{id}_1 * \mathbf{id}_2 \$$	shift
$\$\,\mathbf{id}_1$	$* \mathbf{id}_2 \$$	reduce by $F \rightarrow \mathbf{id}$
$\$\,F$	$* \mathbf{id}_2 \$$	reduce by $T \rightarrow F$
$\$\,T$	$* \mathbf{id}_2 \$$	shift
$\$\,T *$	$\mathbf{id}_2 \$$	shift
$\$\,T * \mathbf{id}_2$	$\$$	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-> 0S1 | 01

So…

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