Grammar

- A 4-tuple $G = <V_N, V_T, P, S>$ of a language L(G)
  - V_N is a set of nonterminal symbols used to write the grammar
  - V_T is the set of terminals (set of words in the language L(G))
  - P is a set of production rules
  - S is a special symbol in V_N, called the start symbol of the grammar
- Strings in language L(G) are those derived from S by applying the production rules from P
- Examples:

  E -> E + T | T
  T -> T * F | F
  F -> (E) | **id**

  E -> TE’
  E’ -> +TE’ | Ɛ
  T -> FT’
  T’ -> *FT’ | Ɛ
  F -> (E) | **id**

Error handling

- Common programming errors
  - Lexical errors
  - Syntactic errors
  - Semantic errors
  - Lexical errors
- Error handler goals
  - 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

Error-recovery strategies

- Panic mode recovery
  - Discard input symbol one at a time until one of designated set of synchronization tokens is found
- Phrase level recovery
  - Replacing a prefix of remaining input by some string that allows the parser to continue
- Error productions
  - Augment the grammar with productions that generate the erroneous constructs
- Global correction
  - Choosing minimal sequence of changes to obtain a globally least-cost correction

Context free grammars

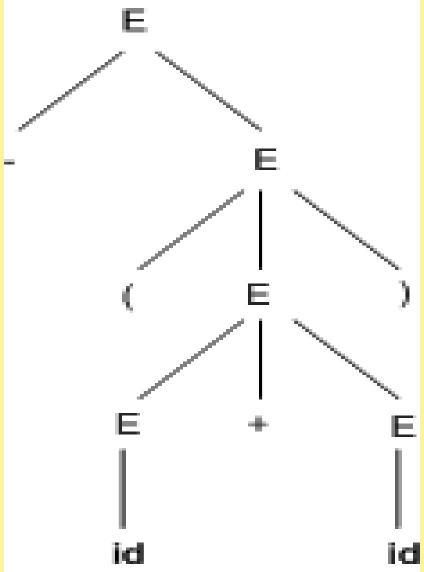
- Terminals
- Nonterminals
- Start symbol
- Productions

expression -> expression + term
expression -> expression – term
expression -> term
term -> term * factor
term -> term / factor
term -> factor
factor -> (expression)
factor -> **id**

Derivations

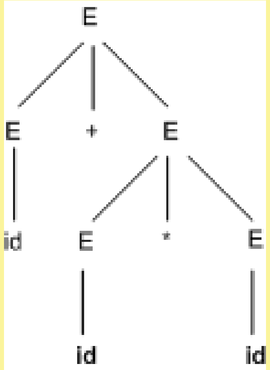
- Productions are treated as rewriting rules to generate a string
- Rightmost and leftmost derivations
  - E -> E + E | E * E | -E | (E) | **id**
  - Derivations for **–(id+id)**
    - E => -E => -(E) => -(E+E) => -(**id**+E)=>-(**id**+**id**)
    - E => -E => -(E) => -(E+E) => -(E+**id**)=>-(**id**+**id**)

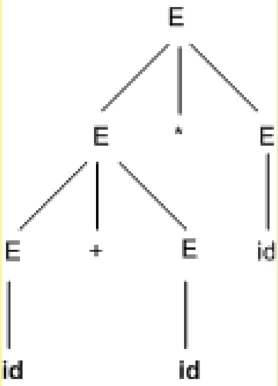
Parse tree



Ambiguity

- For some strings there exist more than one parse tree
- Or more than one leftmost derivation
- Or more than one rightmost derivation
- Example: id+id*id

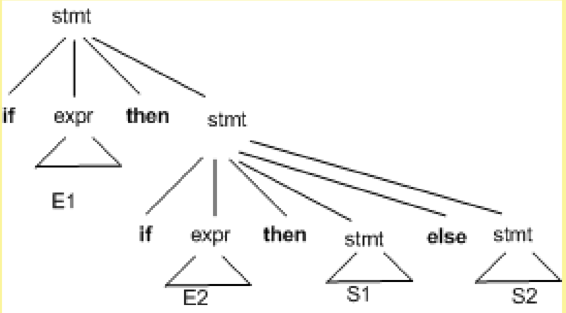




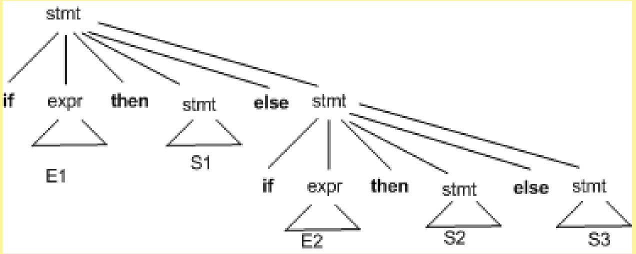
Elimination of ambiguity

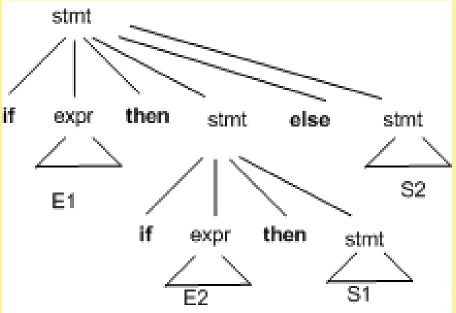
if E1 then if E2 then S1 else S2

stmt ⟶ If expr then stmt
| If expr then stmt else stmt
| other



if E1 then S1 else if E2 then S2 else S3





Elimination of ambiguity (cont.)

- Idea:
  - A statement appearing between a **then** and an **else** must be matched

```
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
```

Elimination of left recursion

- A grammar is left recursive if it has a non-terminal A such that there is a derivation A=> A α
- Top down parsing methods cant handle left-recursive grammars
- A simple rule for direct left recursion elimination:
  - For a rule like:
    - A -> A α | β
  - We may replace it with
    - A -> β A’
    - A’ -> α A’ | ε

Left recursion elimination (cont.)

- There are cases like following
  - S -> Aa | b
  - A -> Ac | Sd | ε
- Left recursion elimination algorithm:
  - Arrange the nonterminals in some order A1,A2,…,An.

  - For (each i from 1 to n) {
    For (each j from 1 to i-1) {
    Replace each production of the form Ai-> Aj γ by the production Ai -> $\delta_1 \gamma$ | $\delta_2 \gamma$ | … | $\delta_k \gamma$ where Aj-> δ_1 | δ_2 | … | δ_k are all current Aj productions
    }
    Eliminate left recursion among the Ai-productions
    }

Left Recursion Elimination Example

E -> E+T | T

T -> T*F | F

F -> (E) | **id**

→

E -> TE’

E’ -> +TE’ | ε

T -> FT’

T’ -> *FT’ | ε

F -> (E) | **id**

Left factoring

- Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive or top-down parsing.
- Consider following grammar:
  - Stmt -> **if** expr **then** stmt **else** stmt
  - | **if** expr **then** stmt
- On seeing input **if** it is not clear for the parser which production to use
- We can easily perform left factoring:
  - If we have A-> $\alpha\ \beta_1 \mid \alpha\ \beta_2$ then we replace it with
    - A -> α A'
    - A' -> $\beta_1 \mid \beta_2$

Left factoring (cont.)

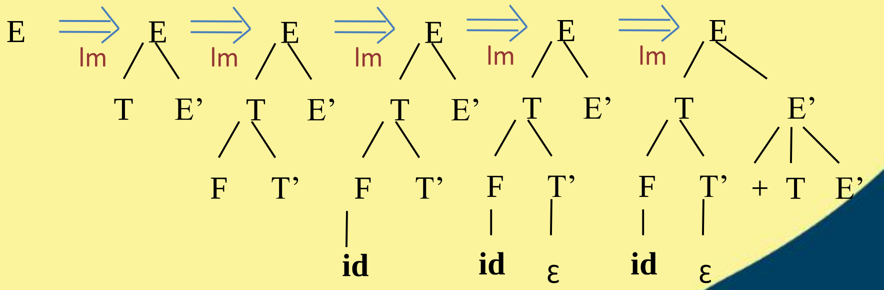
- Algorithm
  - For each non-terminal A, find the longest prefix α common to two or more of its alternatives. If $\alpha \neq \varepsilon$, then replace all of A-productions A-> $\alpha\beta_1 \mid \alpha\beta_2 \mid \ldots \mid \alpha\beta_n \mid \gamma$ by
    - A -> α A' | γ
    - A' -> $\beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$
- Example:
  - S -> i E t S | i E t S e S | a
  - E -> b
- Modifies to
  - S -> i E t S S' | a
  - S' -> e S | ε
  - E -> b

TOP DOWN PARSING

Introduction

- A Top-down parser tries to create a parse tree from the root towards the leafs scanning input from left to right
- It can be also viewed as finding a leftmost derivation for an input string
- Example: id+id*id

E -> TE'
E' -> +TE' | ε
T -> FT'
T' -> *FT' | ε
F -> (E) | **id**



Recursive descent parsing

- Consists of a set of procedures, one for each nonterminal
- Execution begins with the procedure for start symbol
- A typical procedure for a non-terminal

```
void A() {
        choose an A-production, A->X1X2..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 */
        }
}
```

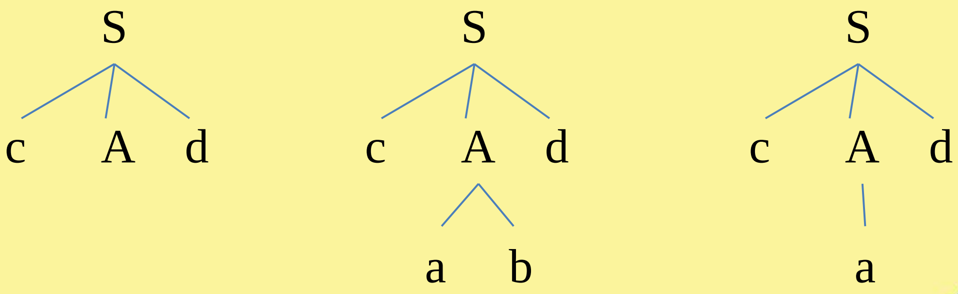
Recursive descent parsing (cont)

- General recursive descent may require backtracking
- The previous code needs to be modified to allow backtracking
- In general form it cannot choose an appropriate production easily.
- So we need to try all alternatives
- If one fails, the input pointer needs to be reset and another alternative has to be tried
- Recursive descent parsers cannot be used for left-recursive grammars

Example

S->cAd
A->ab | a

Input: cad



Predictive parser

- It is a recursive-descent parser that needs no backtracking
- Suppose A -> A1 | A2 | ….| An
- If the non-terminal to be expanded next is ‘A’, then the choice of rule is made on the basis of the current input symbol ‘a’.

Procedure

- Make a transition diagram (like dfa/nfa) for every rule of the grammar.
- Optimize the dfa by reducing the number of states, yielding the final transition diagram
- To parse a string, simulate the string on the transition diagram
- If after consuming the input the transition diagram reaches an accept state, it is parsed.

Example

Consider the grammar:

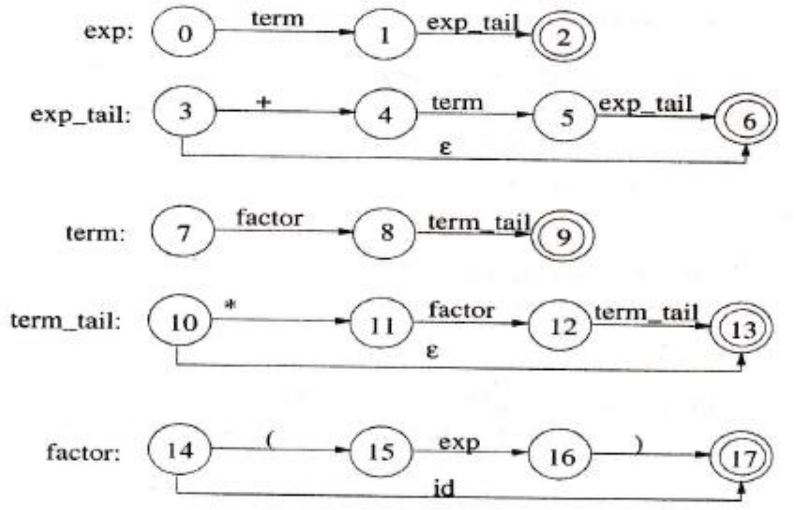
exp -> term exp_tail

exp_tail -> + term exp_tail | ε

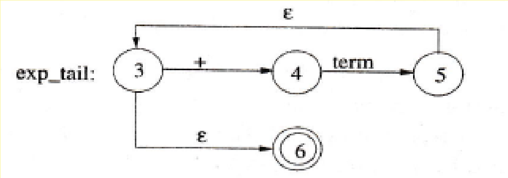
term -> factor term_tail

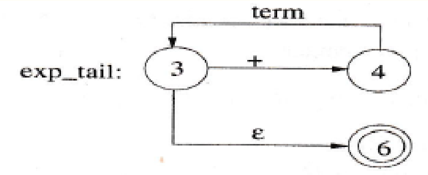
term_tail -> * factor term_tail | ε

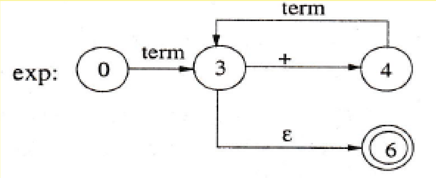
factor -> (exp) | **id**

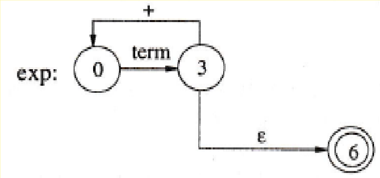


Example – Simplification





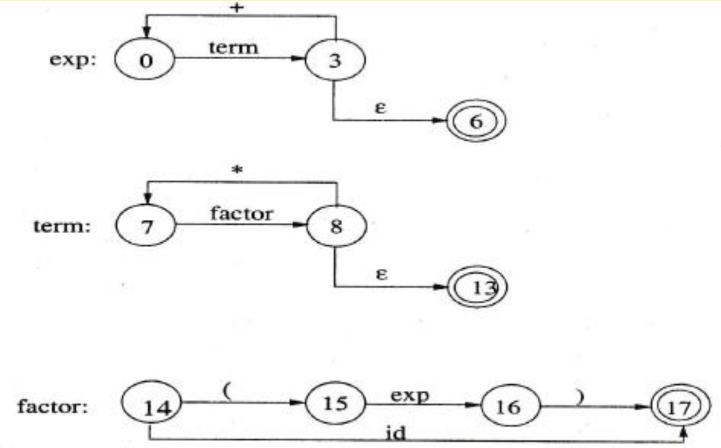




Eliminate self-recursion

Remove redundant ε edge

Substituting exp_tail into exp



Final set of diagrams

SIMULATION METHOD

- Start from the start state
- If a terminal comes consume it, move to next state
- If a non – terminal comes go to the state of the "dfa" of the non-term and return on reaching the final state
- Return to the original "dfa" and continue parsing
- If on completion(reading input string completely), you reach a final state, string is successfully parsed.

Disadvantage

- It is inherently a recursive parser, so it consumes a lot of memory as the stack grows.
- To remove this recursion, we use LL-parser, which uses a table for lookup.

First and Follow

- **First**(α) is set of terminals that begins strings derived from α
- If $\alpha \overset{*}{\Rightarrow} \varepsilon$ then ε is also in First(α)
- In predictive parsing when we have A-> α | β , if First(α) and First(β) are disjoint sets then we can select appropriate A-production by looking at the next input
- **Follow**(A), for any nonterminal A, is set of terminals a that can appear immediately after A in some sentential form
  - If we have $S \overset{*}{\Rightarrow} \alpha Aa \beta$ for some α and β then a is in Follow(A)
- If A can be the rightmost symbol in some sentential form, then $ is in Follow(A)

Computing First

- To compute First(X), apply following rules until no more terminals or ε can be added to any First set:

1. If X is a terminal then First(X) = {X}.

1. If X is a nonterminal and X->$Y_1Y_2 \ldots Y_k$ is a production for some $k \geq 1$, then place a in First(X) if for some i a is in First(Y_i) and ε is in all of First(Y_1),…,First(Y_{i-1}) that is $Y_1 \ldots Y_{i-1} \overset{*}{\Rightarrow} \varepsilon$. If ε is in First(Y_j) for j=1,…,k then add ε to First(X).

1. If X-> ε is a production then add ε to First(X)

Computing Follow

- To compute Follow(A) for all nonterminals A, apply following rules until nothing can be added to any follow set:
  1. Place $ in Follow(S) where S is the start symbol
  2. If there is a production A-> $\alpha \mathrm{B} \beta$ then everything in First(β) except ε is in Follow(B).
  3. If there is a production A-> α B or a production A-> α B β where First(β) contains ε, then everything in Follow(A) is in Follow(B)

Example of First and Follow Sets

E -> TE’
E’ -> +TE’ | ε
T -> FT’
T’ -> *FT’ | ε
F -> (E) | **id**

	First	Follow
F	{(,id}	{+, *,), $}
T	{(,id}	{+,), $}
E	{(,id}	{), $}
E’	{+,ε}	{), $}
T’	{*,ε}	{+,), $}

LL(1) Grammars

- Predictive parsers are those recursive descent parsers needing no backtracking
- Grammars for which we can create predictive parsers are called LL(1)
  - The first L means scanning input from left to right
  - The second L means leftmost derivation
  - And 1 stands for using one input symbol for lookahead
  - More general one is LL(k), with k symbol lookahead
- A grammar G is LL(1) if and only if whenever A-> $\alpha \mid \beta$ are two distinct productions of G, the following conditions hold:
  - For no terminal a do α and β both derive strings beginning with a
  - At most one of α or β can derive empty string
  - If $\alpha \overset{*}{\Rightarrow} \varepsilon$ then β does not derive any string beginning with a terminal in Follow(A)

Construction of predictive parsing table

- For each production A-> α in grammar do the following:
  1. For each terminal a in First(α) add A-> α in M[A,a]
  2. If ε is in First(α), then for each terminal b in Follow(A) add A-> ε to M[A,b]. If ε is in First(α) and $ is in Follow(A), add A-> ε to M[A,$] as well
- If after performing the above, there is no production in M[A,a] then set M[A,a] to error

Example

E -> TE’
E’ -> +TE’ | ε
T -> FT’
T’ -> *FT’ | ε
F -> (E) | **id**

	First	Follow
F	{(,id}	{+, *,), $}
T	{(,id}	{+,), $}
E	{(,id}	{), $}
E’	{+,ε}	{), $}
T’	{*,ε}	{+,), $}

Non - terminal	Input Symbol					
	id	+	*	(	)	$
E	E -> TE’			E -> TE’		
E’		E’ -> +TE’			E’ -> ε	E’ -> ε
T	T -> FT’			T -> FT’		
T’		T’ -> ε	T’ -> *FT’		T’ -> ε	T’ -> ε
F	F -> **id**			F -> (E)		