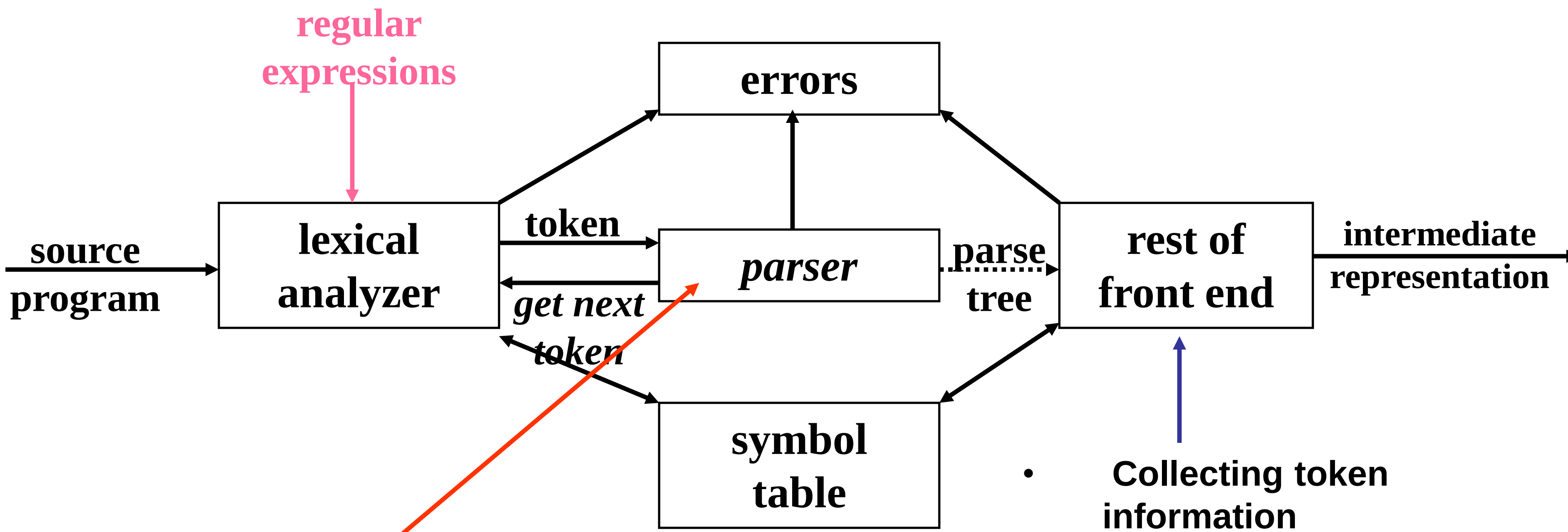
CSE 4102
Syntax Analysis
Or
Parsing

Lecture 04

Parsing

- A.K.A. Syntax Analysis
  - Recognize sentences in a language.
  - Discover the structure of a document/program.
  - Construct (implicitly or explicitly) a tree (called as a parse tree) to represent the structure.
  - The above tree is used later to guide translation.

Parsing During Compilation



- Collecting token information
- Perform type checking
- Intermediate code generation

- Uses a grammar to check structure of tokens
- Produces a parse tree
- Syntactic errors and recovery
- Recognize correct syntax
- Report errors

Parsing Responsibilities

Syntax Error Identification / Handling

Recall typical error types:

1. **Lexical : Misspellings**

   if x<1 thenn y = 5:

2. **Syntactic : Omission, wrong order of tokens**

   if ((x<1) & (y>5)))

3. **Semantic : Incompatible types, undefined IDs**

   if (x+5) then

4. **Logical : Infinite loop / recursive call**

   if (i<9) then ...
   Should be <= not <

Majority of error processing occurs during syntax analysis

NOTE: Not all errors are identifiable !!

Error Detection

- **Much responsibility on Parser**
  - Many errors are syntactic in nature
  - Modern parsing method can detect the presence of syntactic errors in programs very efficiently
  - Detecting semantic or logical error is difficult

- Challenges for error handler in Parser
  - It should report error clearly and accurately
  - It should **recover from error and continue**..
  - It should **not significantly slow down** the processing of correct programs

- Good news is
  - Common errors are simple and relatively easy to catch.

- Errors don't occur that frequently!!
    - 60% programs are syntactically and semantically correct
    - 80% erroneous statements have only 1 error, 13% have 2
    - Most error are trivial : 90% single token error
    - 60% punctuation, 20% operator, 15% keyword, 5% other error

Adequate Error Reporting is Not a Trivial Task

- Difficult to generate clear and accurate error messages.

Example

```
function foo () {
...
if (...) {
...
} else {
   ...
```

Missing } here

```
...
}
<eof>
```

Not detected until here

Example

```
int myVarr;
...
x = myVar;
...
```

Misspelled ID here

Not detected until here

Error Recovery

- After first error recovered
  - Compiler must go on!
    - Restore to some state and process the rest of the input

- **Error-Correcting Compilers**
  - Issue an error message
  - Fix the problem
  - Produce an executable

Example

```
Error on line 23: "myVarr" undefined.
"myVar" was used.
```

May not be a good Idea!!

- Guessing the programmers intention is not easy!

Error Recovery May Trigger More Errors!

- Inadequate recovery may introduce more errors
  - Those were not programmers errors

- Example:

```
int myVar flag ;
...
x := flag;
...
...
while (flag==0)
...
```

Declaration of flag is discarded

Variable flag is undefined

Variable flag is undefined

Too many Error message may be obscuring

- May bury the real message
- Remedy:
  - allow 1 message per token or per statement
  - Quit after a maximum (e.g. 100) number of errors

Error Recovery Approaches: Panic Mode

- Discard tokens until we see a “synchronizing” token.

Example

Skip to next occurrence of

`} end ;`

Resume by parsing the next statement

- The key...
  - Good set of synchronizing tokens
  - Knowing what to do then
- Advantage
  - Simple to implement
  - Does not go into infinite loop
  - Commonly used
- Disadvantage
  - May skip over large sections of source with some errors

Error Recovery Approaches: Phrase-Level Recovery

- **Compiler corrects the program**

  by deleting or inserting tokens

  ...so it can proceed to parse from where it was.

Example

while (x==4)   y:= a + b

Insert do to fix the statement

- The key...

  Don’t get into an infinite loop

Context Free Grammars (CFG)

- A **context free grammar** is a formal model that consists of:
- **Terminals**

  Keywords

  Token Classes

  Punctuation
- **Non-terminals**

  Any symbol appearing on the lefthand side of any rule
- **Start Symbol**

  Usually the non-terminal on the lefthand side of the first rule
- **Rules (or “Productions”)**

  BNF: Backus-Naur Form / Backus-Normal Form

  Stmt **::= if** Expr **then** Stmt **else** Stmt

Rule Alternative Notations

$E \rightarrow E + E$
$E \rightarrow (E)$
$E \rightarrow - E$
$E \rightarrow ID$

$E \rightarrow E + E$
$\rightarrow (E)$
$\rightarrow - E$
$\rightarrow ID$

$E \rightarrow E + E$
$\mid (E)$
$\mid - E$
$\mid ID$

$E \rightarrow E + E \mid (E) \mid - E \mid ID$

All Notations are Equivalent

Context Free Grammars : A First Look

assign_stmt → *id := expr ;*

expr → *expr operator term*

expr → *term*

term → *id*

term → *real*

term → *integer*

operator → *+*

operator → *-*

Derivation: A sequence of grammar rule applications and substitutions that transform a starting non-term into a sequence of terminals / tokens.

Derivation

Let’s derive: *id := id + real – integer ;* **using production:**

assign_stmt	*assign_stmt* $\rightarrow$ *id := expr ;*
$\rightarrow$ *id := expr ;*	*expr* $\rightarrow$ *expr operator term*
$\rightarrow$*id := expr operator term;*	*expr* $\rightarrow$ *expr operator term*
$\rightarrow$*id := expr operator term operator term;*	*expr* $\rightarrow$ *term*
$\rightarrow$ *id := term operator term operator term;*	*term* $\rightarrow$ *id*
$\rightarrow$ *id := id operator term operator term;*	*operator* $\rightarrow$ +
$\rightarrow$ *id := id + term operator term;*	*term* $\rightarrow$ *real*
$\rightarrow$ *id := id + real operator term;*	*operator* $\rightarrow$ -
$\rightarrow$ *id := id + real - term;*	*term* $\rightarrow$ *integer*
$\rightarrow$ *id := id + real - integer;*	

Example Grammar: Simple Arithmetic Expressions

$$expr \rightarrow expr \;\; op \;\; expr$$
$$expr \rightarrow (\; expr \;)$$
$$expr \rightarrow - \; expr$$
$$expr \rightarrow id$$
$$op \rightarrow +$$
$$op \rightarrow -$$
$$op \rightarrow *$$
$$op \rightarrow /$$
$$op \rightarrow \uparrow$$

9 Production rules

Terminals: id + - * / ↑ ( )
Nonterminals: *expr, op*
Start symbol: *expr*

Notational Conventions

- Terminals
  - Lower-case letters early in the alphabet: *a, b, c*
  - Operator symbols: +, -
  - Punctuations symbols: parentheses, comma
  - Boldface strings: **id** or **if**

- Nonterminals:
  - Upper-case letters early in the alphabet: *A, B, C*
  - The letter *S* (start symbol)
  - Lower-case italic names: *expr* or *stmt*

- Upper-case letters late in the alphabet, such as *X, Y, Z,* represent either nonterminals or terminals.

- Lower-case letters late in the alphabet, such as *u, v, …, z,* represent strings of terminals.

Notational Conventions

- Lower-case Greek letters, such as α, β, γ, represent strings of grammar symbols. Thus $A \rightarrow \alpha$ indicates that there is a single nonterminal A on the left side of the production and a string of grammar symbols α to the right of the arrow.

- If $A \rightarrow \alpha_1$, $A \rightarrow \alpha_2$, …., $A \rightarrow \alpha_k$ are all productions with A on the left, we may write $A \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots. \mid \alpha_k$

- Unless otherwise started, the left side of the first production is the start symbol.

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

$$A \rightarrow + \mid - \mid * \mid / \mid \uparrow$$

Derivations

1. $E \rightarrow E + E$
2. $ \rightarrow E * E$
3. $ \rightarrow (E)$
4. $ \rightarrow - E$
5. $ \rightarrow$ ID

A "Derivation" of "(id*id)"

$$E \Rightarrow (E) \Rightarrow (E*E) \Rightarrow \underbrace{(id*E)} \Rightarrow (id*id)$$

"Sentential Forms"

Doesn't contain nonterminals

Derivation

If $A \rightarrow \beta$ is a rule, then we can write

$$\underbrace{\alpha A \gamma} \Rightarrow \alpha \beta \gamma$$

Any sentential form containing a nonterminal (call it A*)*

... such that A *matches the nonterminal in some rule.*

Derives in zero-or-more steps $\Rightarrow^*$

$$E \Rightarrow^* (\underline{\mathtt{id}} * \underline{\mathtt{id}})$$

If $\alpha \Rightarrow^* \beta$ and $\beta \Rightarrow \gamma$, then $\alpha \Rightarrow^* \gamma$

Derives in one-or-more steps $\Rightarrow^+$

Given

G A grammar

S The Start Symbol

Define

L(G) The language generated

$L(G) = \{ w \mid S \Rightarrow^{+} w \}$

"Equivalence" of CFG's

If two CFG's generate the same language, we say they are "equivalent."

$G_1 \approx G_2$ whenever $L(G_1) = L(G_2)$

In making a derivation...

Choose which nonterminal to expand

Choose which rule to apply

Leftmost Derivation

In a derivation... always expand the *leftmost* nonterminal.

$$
\begin{aligned}
& E \\
\Rightarrow\ & E+E \\
\Rightarrow\ & (E)+E \\
\Rightarrow\ & (E*E)+E \\
\Rightarrow\ & (\underline{\mathtt{id}}*E)+E \\
\Rightarrow\ & (\underline{\mathtt{id}}*\underline{\mathtt{id}})+E \\
\Rightarrow\ & (\underline{\mathtt{id}}*\underline{\mathtt{id}})+\underline{\mathtt{id}}
\end{aligned}
$$

1.	E	$\rightarrow$ E + E
2.		$\rightarrow$ E * E
3.		$\rightarrow$ (E)
4.		$\rightarrow$ - E
5.		$\rightarrow$ ID

Let $\Rightarrow_{LM}$ denote a step in a leftmost derivation ($\Rightarrow_{LM}{}^{*}$ means zero-or-more steps)

At each step in a leftmost derivation, we have

$w A \gamma \Rightarrow_{LM} w \beta \gamma$ where $A \rightarrow \beta$ is a rule

(Recall that w is a string of terminals.)

Each sentential form in a leftmost derivation is called a "left-sentential form."

If $S \Rightarrow_{LM}{}^{*} \alpha$ then we say α is a "left-sentential form."

Rightmost Derivation

In a derivation... always expand the *rightmost* nonterminal.

$$
\begin{aligned}
& E \\
\Rightarrow\ & E+E \\
\Rightarrow\ & E+\underline{id} \\
\Rightarrow\ & (E)+\underline{id} \\
\Rightarrow\ & (E*E)+\underline{id} \\
\Rightarrow\ & (E*\underline{id})+\underline{id} \\
\Rightarrow\ & (\underline{id}*\underline{id})+\underline{id}
\end{aligned}
$$

1.	E	$\rightarrow$ E + E
2.		$\rightarrow$ E * E
3.		$\rightarrow$ (E)
4.		$\rightarrow$ - E
5.		$\rightarrow$ ID

Let $\Rightarrow_{RM}$ denote a step in a rightmost derivation ($\Rightarrow_{RM}{}^{*}$ means zero-or-more steps)

At each step in a rightmost derivation, we have

$\alpha A w \Rightarrow_{RM} \alpha\beta w$ where $A \rightarrow \beta$ is a rule

(Recall that w is a string of terminals.)

Each sentential form in a rightmost derivation is called a "right-sentential form."

If $S \Rightarrow_{RM}{}^{*} \alpha$ then we say α is a "right-sentential form."

Parse Tree

Two choices at each step in a derivation...

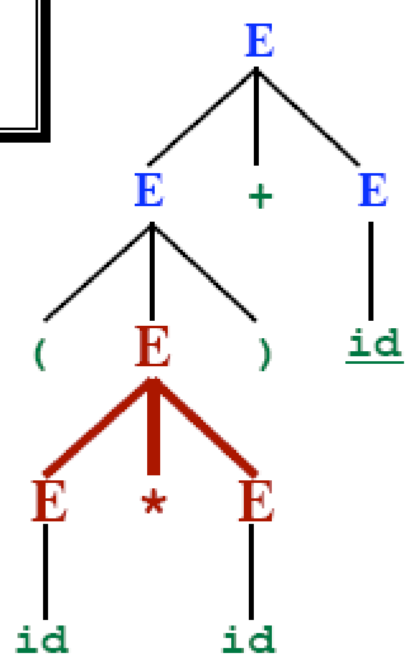
- Which non-terminal to expand
- Which rule to use in replacing it

The parse tree remembers only this

Leftmost Derivation:

$$
\begin{array}{ll}
 & E \\
\Rightarrow & E+E \\
\Rightarrow & (E)+E \\
\Rightarrow & (E*E)+E \\
\Rightarrow & (\underline{id}*E)+E \\
\Rightarrow & (\underline{id}*\underline{id})+E \\
\Rightarrow & (\underline{id}*\underline{id})+\underline{id}
\end{array}
$$

1.	E	$\rightarrow$ E + E
2.		$\rightarrow$ E * E
3.		$\rightarrow$ (E)
4.		$\rightarrow$ - E
5.		$\rightarrow$ ID



Parse Tree

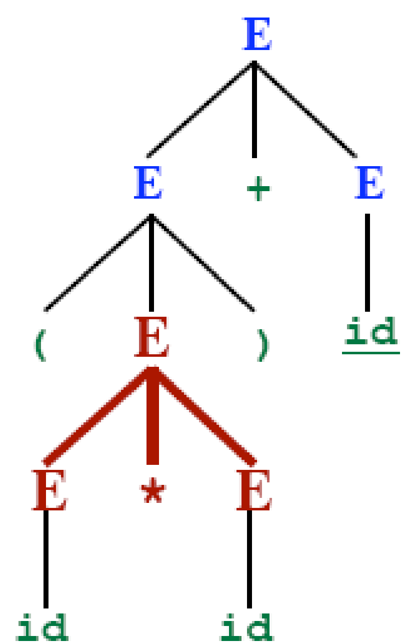
Two choices at each step in a derivation...

- Which non-terminal to expand
- Which rule to use in replacing it

The parse tree remembers only this

Rightmost Derivation:

E
⇒ E+E
⇒ E+id
⇒ (E)+id
⇒ (E*E)+id
⇒ (E*id)+id
⇒ (id*id)+id



1.	E	→	E + E
2.		→	E * E
3.		→	(E)
4.		→	- E
5.		→	ID

Parse Tree

Two choices at each step in a derivation…

- Which non-terminal to expand
- Which rule to use in replacing it

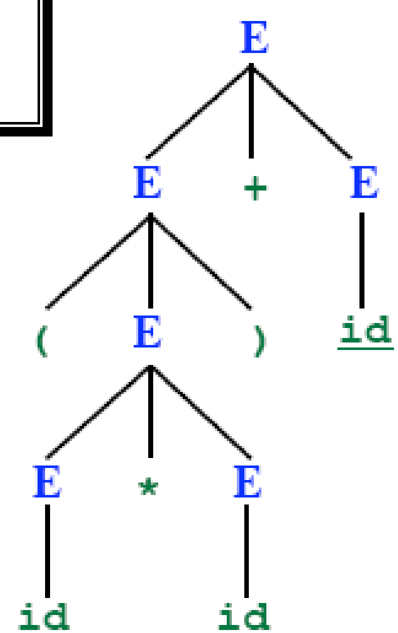
The parse tree remembers only this

Leftmost Derivation:

E
⇒ E+E
⇒ (E)+E
⇒ (E*E)+E
⇒ (id*E)+E
⇒ (id*id)+E
⇒ (id*id)+id

Rightmost Derivation:

E
⇒ E+E
⇒ E+id
⇒ (E)+id
⇒ (E*E)+id
⇒ (E*id)+id
⇒ (id*id)+id



1. E → E + E
2. → E * E
3. → (E)
4. → - E
5. → ID

Parse Tree

Given a leftmost derivation, we can build a parse tree.
Given a rightmost derivation, we can build a parse tree.

Lefttmost Derivation of
`(id*id)+id`

Rightmost Derivation of
`(id*id)+id`

Same Parse Tree

Every parse tree corresponds to...

- A single, unique leftmost derivation
- A single, unique rightmost derivation

Ambiguity:

However, one input string may have several parse trees!!!

Therefore:

- Several leftmost derivations
- Several rightmost derivations

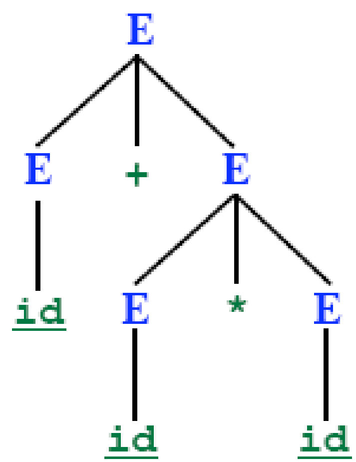
Ambiguous Grammar

1.	E	→	E + E
2.		→	E * E
3.		→	(E)
4.		→	- E
5.		→	ID

Input: `id+id*id`

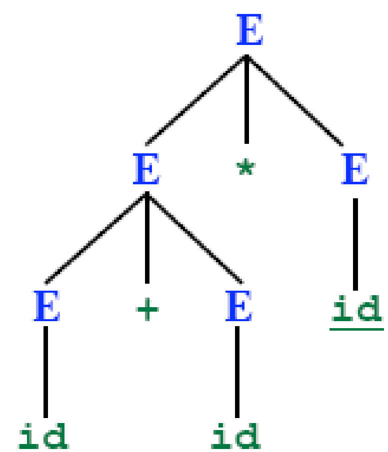
Leftmost Derivation #1

E
⇒ E+E
⇒ id+E
⇒ id+E*E
⇒ id+id*E
⇒ id+id*id



Leftmost Derivation #2

E
⇒ E*E
⇒ E+E*E
⇒ id+E*E
⇒ id+id*E
⇒ id+id*id



Ambiguous Grammar

- More than one Parse Tree for some sentence.
  - The grammar for a programming language may be ambiguous
  - Need to modify it for parsing.

- Also: Grammar may be left recursive.
- Need to modify it for parsing.

Elimination of Ambiguity

- Ambiguous
- A Grammar is ambiguous if there are multiple parse trees for the same sentence.
- Disambiguation
- Express Preference for one parse tree over others
  - Add disambiguating rule into the grammar

Removing Ambiguity

Take Original Grammar:

stmt* → if *expr* then *stmt
| if *expr* then *stmt* else *stmt*
| other (any other statement)

Rule: Match each **else** with the closest previous unmatched **then**.

Revise to remove ambiguity:

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

Resolving Difficulties : Left Recursion

A left recursive grammar has rules that support the derivation : $A \overset{+}{\Rightarrow} A\alpha$, for some α.

Top-Down parsing can't reconcile this type of grammar, since it could consistently make choice which wouldn't allow termination.

$$A \Rightarrow A\alpha \Rightarrow A\alpha\alpha \Rightarrow A\alpha\alpha\alpha \ldots \text{ etc.} \quad A \rightarrow A\alpha \mid \beta$$

Take left recursive grammar:

$$A \rightarrow A\alpha \mid \beta$$

To the following:

$$A \rightarrow \beta A'$$

$$A' \rightarrow \alpha A' \mid \in$$

Why is Left Recursion a Problem ?

Consider:

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

Derive : id + id + id

$$E \Rightarrow E + T \Rightarrow$$

How can left recursion be removed ?

$$E \rightarrow E + T \mid T$$

What does this generate?

$$E \Rightarrow E + T \Rightarrow T + T$$

$$E \Rightarrow E + T \Rightarrow E + T + T \Rightarrow T + T + T$$

…

How does this build strings ?

What does each string have to start with ?

Resolving Difficulties : Left Recursion (2)

Informal Discussion:

Take all productions for A and order as:

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

Where no β_i begins with A.

Now apply concepts of previous slide:

$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \ldots \mid \beta_n A'$

$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \ldots \mid \alpha_m A' \mid \epsilon$

For our example:

$E \rightarrow E + T \mid T$ $\longrightarrow$ $\left\{ \begin{array}{l} E \rightarrow TE' \\ E' \rightarrow + TE' \mid \epsilon \end{array} \right.$

$T \rightarrow T * F \mid F$ $\longrightarrow$ $\left\{ \begin{array}{l} T \rightarrow FT' \\ T' \rightarrow * FT' \mid \epsilon \end{array} \right.$

$F \rightarrow (E) \mid id$ $\longrightarrow$ $F \rightarrow (E) \mid id$

Resolving Difficulties : Left Recursion (3)

Problem: If left recursion is two-or-more levels deep, this isn't enough

$$\left.\begin{array}{l} S \rightarrow Aa \mid b \\ A \rightarrow Ac \mid Sd \mid \epsilon \end{array}\right\} \qquad S \Rightarrow Aa \Rightarrow Sda$$

Algorithm:

Input: **Grammar G with ordered Non-Terminals $A_1, ..., A_n$**

Output: **An equivalent grammar with no left recursion**

1. **Arrange the non-terminals in some order A_1=start NT, $A_2, \ldots A_n$**
2. `for i := 1 to n  do begin`

    `for j := 1 to i - 1  do begin`

    replace each production of the form $A_i \rightarrow A_j\gamma$
    by the productions $A_i \rightarrow \delta_1\gamma \mid \delta_2\gamma \mid \ldots \mid \delta_k\gamma$
    where $A_j \rightarrow \delta_1|\delta_2|\ldots|\delta_k$ are all current A_j productions`;`

    `end`

    eliminate the immediate left recursion among A_i productions

  `end`

Using the Algorithm

Apply the algorithm to: $A_1 \rightarrow A_2a \mid b \mid \in$

$A_2 \rightarrow A_2c \mid A_1d$

i = 1

For A_1 there is no left recursion

i = 2

for j=1 to 1 do

Take productions: $A_2 \rightarrow A_1\gamma$ **and replace with**

$A_2 \rightarrow \delta_1 \gamma \mid \delta_2 \gamma \mid \dots \mid \delta_k \gamma \mid$

where $A_1 \rightarrow \delta_1 \mid \delta_2 \mid \dots \mid \delta_k$ **are A_1 productions**

in our case $A_2 \rightarrow A_1d$ **becomes** $A_2 \rightarrow A_2ad \mid bd \mid d$

What's left: $A_1 \rightarrow A_2a \mid b \mid \in$

$A_2 \rightarrow A_2c \mid A_2ad \mid bd \mid d$

Are we done ?

Using the Algorithm (2)

No ! We must still remove A_2 left recursion !

$$A_1 \to A_2 a \mid b \mid \in$$

$$A_2 \to A_2\, c \mid A_2\, ad \mid bd \mid d$$

Recall:

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

$$A \to \beta_1 A' \mid \beta_2 A' \mid \ldots \mid \beta_n A'$$

$$A' \to \alpha_1 A' \mid \alpha_2 A' \mid \ldots \mid \alpha_m A' \mid \in$$

$$A_1 \to A_2 a \mid b \mid \in$$

$$A_2 \to bdA_2' \mid dA_2'$$

$$A_2' \to c\, A_2' \mid adA_2' \mid \in$$

Apply to above case. What do you get ?

Removing Difficulties : $\in$-Moves

Transformation: **In order to remove $A \rightarrow \in$ find all rules of the form $B \rightarrow uAv$ and *add* the rule $B \rightarrow uv$ to the grammar G.**

Why does this work ?

Examples:

$$E \rightarrow TE'$$
$$E' \rightarrow + TE' \mid \in$$
$$T \rightarrow FT'$$
$$T' \rightarrow * FT' \mid \in$$
$$F \rightarrow (E) \mid id$$

$$A_1 \rightarrow A_2\ a \mid b$$
$$A_2 \rightarrow bd\ A_2' \mid A_2'$$
$$A_2' \rightarrow c\ A_2' \mid bd\ A_2' \mid \in$$

A is Grammar $\in$-free if:

1. **It has no $\in$-production or**
2. **There is exactly one $\in$-production**

   $S \rightarrow \in$ and then the start symbol S does not appear on the right side of any production.

Removing Difficulties : Left Factoring

Problem : Uncertain which of 2 rules to choose:

$$stmt \rightarrow \textbf{if}\ expr\ \textbf{then}\ stmt\ \textbf{else}\ stmt$$

$$\mid \textbf{if}\ expr\ \textbf{then}\ stmt$$

When do you know which one is valid ?

What's the general form of *stmt* ?

$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$

α : **if** *expr* **then** *stmt*

β_1: **else** *stmt* β_2 : $\in$

Transform to:

$A \rightarrow \alpha\ A'$

$A' \rightarrow \beta_1 \mid \beta_2$

EXAMPLE:

$stmt \rightarrow \textbf{if}\ expr\ \textbf{then}\ stmt\ rest$

$rest \rightarrow \textbf{else}\ stmt \mid \in$

Top Down Parsing

Top Down Parsing

- Find a left-most derivation
- Find (build) a parse tree
- Start building from the root and work down…
- As we search for a derivation
  – Must make choices:
    - Which rule to use
    - Where to use it

- May run into problems!!

Top-Down Parsing

- Recursive-Descent Parsing
  - Backtracking is needed (If a choice of a production rule does not work, we backtrack to try other alternatives.)
  - It is a general parsing technique, but not widely used.
  - Not efficient
- Predictive Parsing
  - no backtracking
  - efficient
  - needs a special form of grammars (LL(1) grammars).
  - Recursive Predictive Parsing is a special form of Recursive Descent parsing without backtracking.
  - Non-Recursive (Table Driven) Predictive Parser is also known as LL(1) parser.

Recursive Descent Parsing (Backtracking)

Input: `aabbde`

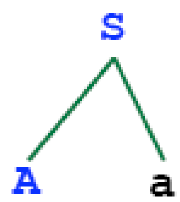
↑

S

1.	S	→	Aa
2.		→	Ce
3.	A	→	aaB
4.		→	aaba
5.	B	→	bbb
6.	C	→	aaD
7.	D	→	bbd

Recursive Descent Parsing (Backtracking)

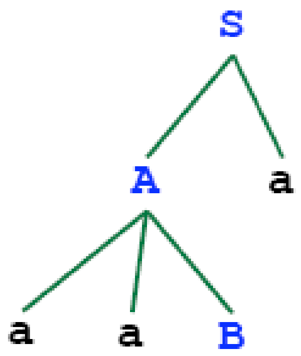
Input: aabbde



1. S → Aa
2. → Ce
3. A → aaB
4. → aaba
5. B → bbb
6. C → aaD
7. D → bbd

Recursive Descent Parsing (Backtracking)

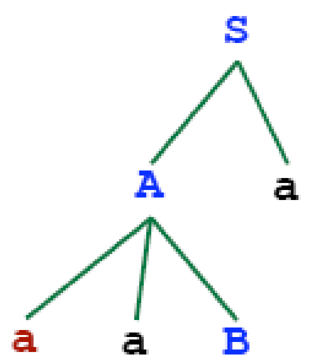
Input: aabbde



1. S → Aa
2. → Ce
3. A → aaB
4. → aaba
5. B → bbb
6. C → aaD
7. D → bbd

Recursive Descent Parsing (Backtracking)

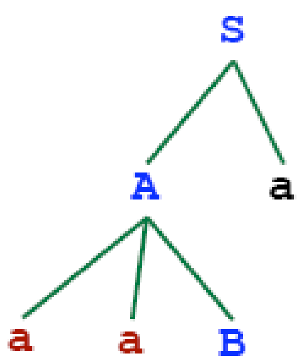
Input: aabbde



1. S → Aa
2. → Ce
3. A → aaB
4. → aaba
5. B → bbb
6. C → aaD
7. D → bbd

Recursive Descent Parsing (Backtracking)

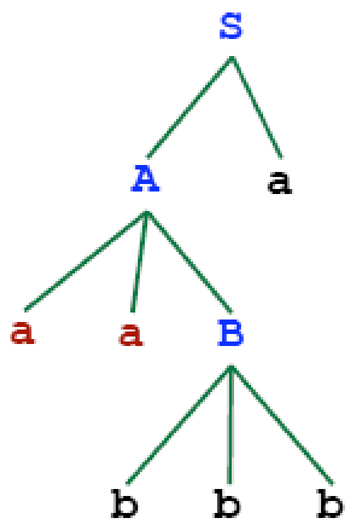
Input: `aabbde`



1. S → A`a`
2. → C`e`
3. A → `aa`B
4. → `aaba`
5. B → `bbb`
6. C → `aa`D
7. D → `bbd`

Recursive Descent Parsing (Backtracking)

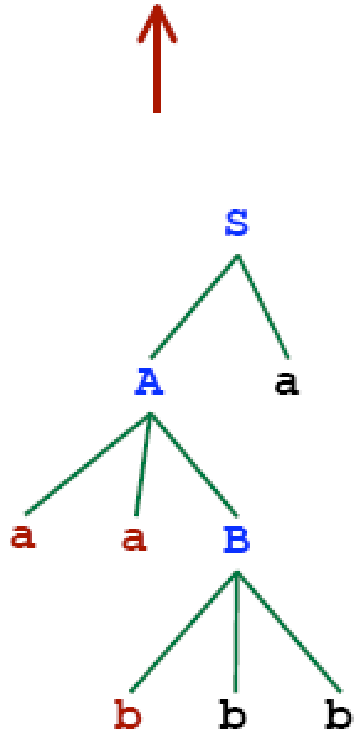
Input: aabbde



1. S → Aa
2. → Ce
3. A → aaB
4. → aaba
5. B → bbb
6. C → aaD
7. D → bbd

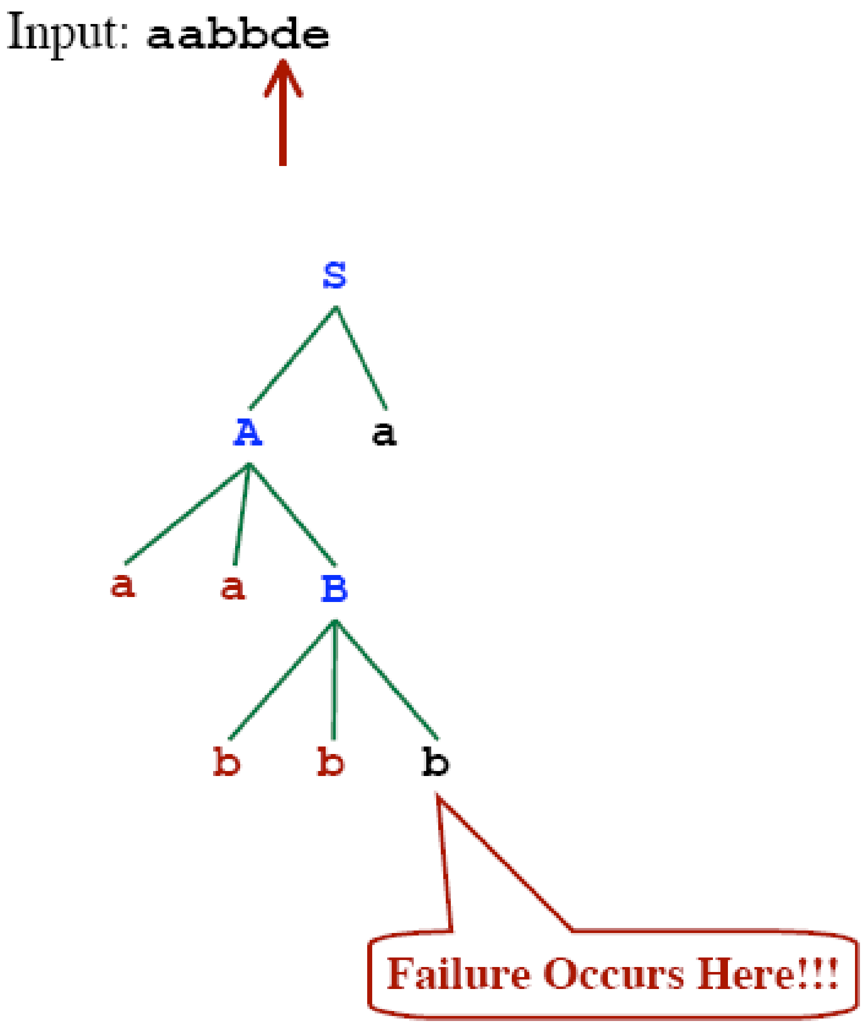
Recursive Descent Parsing (Backtracking)

Input: aabbde



1.	S	$\rightarrow$	Aa
2.		$\rightarrow$	Ce
3.	A	$\rightarrow$	aaB
4.		$\rightarrow$	aaba
5.	B	$\rightarrow$	bbb
6.	C	$\rightarrow$	aaD
7.	D	$\rightarrow$	bbd

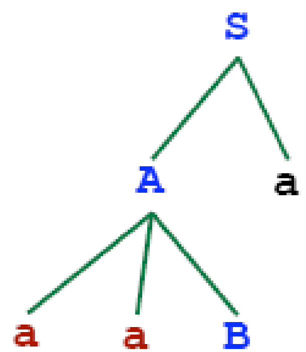
Recursive Descent Parsing (Backtracking)



1. S → Aa
2. → Ce
3. A → aaB
4. → aaba
5. B → bbb
6. C → aaD
7. D → bbd

Recursive Descent Parsing (Backtracking)

Input: aabbde

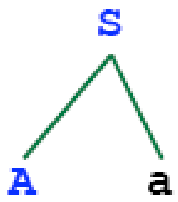


1. S → Aa
2. → Ce
3. A → aaB
4. → aaba
5. B → bbb
6. C → aaD
7. D → bbd

We need an ability to back up in the input!!!

Recursive Descent Parsing (Backtracking)

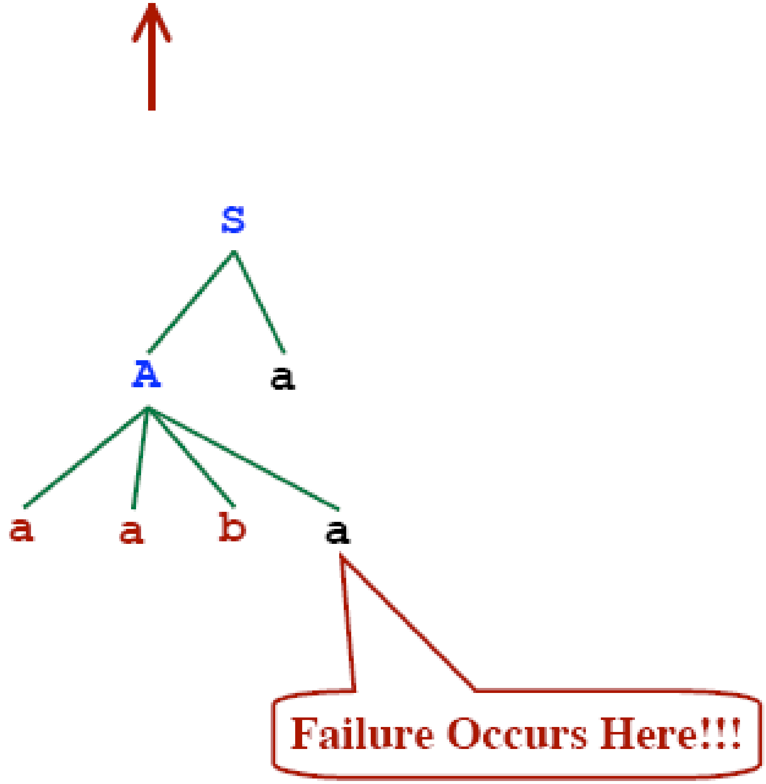
Input: `aabbde`



1. S → Aa
2. → Ce
3. A → aaB
4. → aaba
5. B → bbb
6. C → aaD
7. D → bbd

Recursive Descent Parsing (Backtracking)

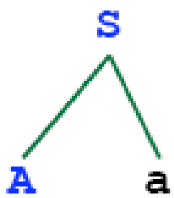
Input: aabbde



1. S → Aa
2. → Ce
3. A → aaB
4. → aaba
5. B → bbb
6. C → aaD
7. D → bbd

Recursive Descent Parsing (Backtracking)

Input: aabbde



1. S $\rightarrow$ Aa
2. $\rightarrow$ Ce
3. A $\rightarrow$ aaB
4. $\rightarrow$ aaba
5. B $\rightarrow$ bbb
6. C $\rightarrow$ aaD
7. D $\rightarrow$ bbd

Recursive Descent Parsing (Backtracking)

Input: aabbde

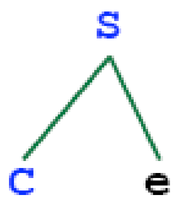
↑

S

1. S → Aa
2. → Ce
3. A → aaB
4. → aaba
5. B → bbb
6. C → aaD
7. D → bbd

Recursive Descent Parsing (Backtracking)

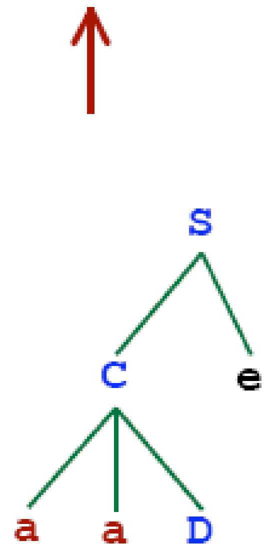
Input: `aabbde`



1. S → Aa
2. → Ce
3. A → aaB
4. → aaba
5. B → bbb
6. C → aaD
7. D → bbd

Recursive Descent Parsing (Backtracking)

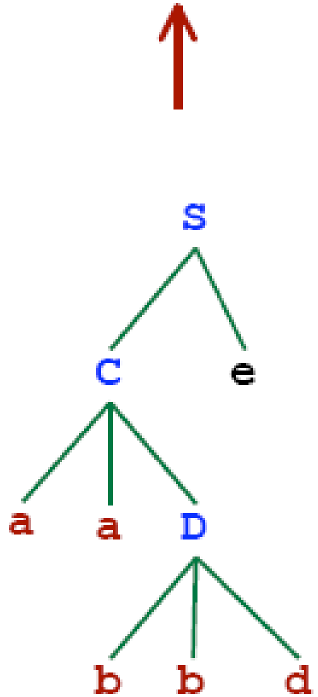
Input: aabbde



1. S → Aa
2. → Ce
3. A → aaB
4. → aaba
5. B → bbb
6. C → aaD
7. D → bbd

Recursive Descent Parsing (Backtracking)

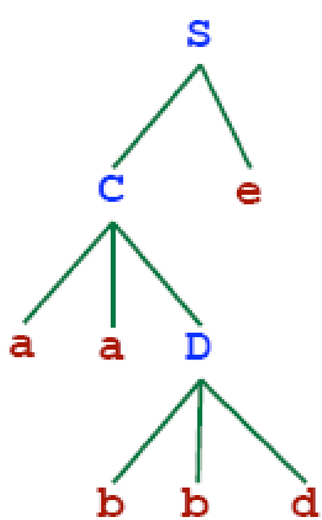
Input: aabbde



1. S → Aa
2. → Ce
3. A → aaB
4. → aaba
5. B → bbb
6. C → aaD
7. D → bbd

Recursive Descent Parsing (Backtracking)

Input: `aabbde`



1.	S	→	Aa
2.		→	Ce
3.	A	→	aaB
4.		→	aaba
5.	B	→	bbb
6.	C	→	aaD
7.	D	→	bbd

Successfully parsed!!

Recursive-Descent Parsing Algorithm

- A recursive-descent parsing program consists of a set of procedures – one for each non-terminal
- Execution begins with the procedure for the start symbol
  - Announces success if the procedure body scans the entire input

```
void A(){
    for (j=1 to t){ /* assume there is t number of A-productions */
        Choose a A-production, Aj→X1X2…Xk;
        for (i=1 to k){
            if (Xi is a non-terminal)
                call procedure Xi();
            else if (Xi equals the current input symbol a)
                advance the input to the next symbol;
            else backtrack in input and reset the pointer
        }
    }
}
```

Predictive Parser

When re-writing a non-terminal in a derivation step, a predictive parser can uniquely choose a production rule by just looking the current symbol in the input string.

$A \rightarrow \alpha_1 \mid \ldots \mid \alpha_n$ input: ... a ....

current token

Predictive Parser (example)

stmt → `if` ...... |
`while` ...... |
`begin` ...... |
`for` .....

- When we are trying to write the non-terminal *stmt*, if the current token is `if` we have to choose first production rule.
- When we are trying to write the non-terminal *stmt*, we can uniquely choose the production rule by just looking the current token.
- We eliminate the left recursion in the grammar, and left factor it. But it may not be suitable for predictive parsing (not LL(1) grammar).

Recursive Predictive Parsing

- Each non-terminal corresponds to a procedure.

Ex: A $\rightarrow$ aBb (This is only the production rule for A)

proc A {

- match the current token with a, and move to the next token;
- call 'B';
- match the current token with b, and move to the next token;

}

Recursive Predictive Parsing (cont.)

$A \rightarrow aBb \mid bAB$

proc A {

case of the current token {

‘a’: - match the current token with a, and move to the next token;

- call ‘B’;
- match the current token with b, and move to the next token;

‘b’: - match the current token with b, and move to the next token;

- call ‘A’;
- call ‘B’;

}

}

Recursive Predictive Parsing (cont.)

- When to apply ε-productions.

  $A \rightarrow aA \mid bB \mid \varepsilon$

- If all other productions fail, we should apply an ε-production. For example, if the current token is not a or b, we may apply the ε-production.
- Most correct choice: We should apply an ε-production for a non-terminal A when the current token is in the follow set of A (which terminals can follow A in the sentential forms).

First Function

Let α be a string of symbols (terminals and nonterminals)

Define:

FIRST (α) = The set of terminals that could occur first in any string derivable from α

$= \{ a \mid \alpha \Rightarrow^* aw, \text{ plus } \varepsilon \text{ if } \alpha \Rightarrow^* \varepsilon \}$

Example:

$E \rightarrow T\ E'$
$E' \rightarrow +\ T\ E' \mid \varepsilon$
$T \rightarrow F\ T'$
$T' \rightarrow *\ F\ T' \mid \varepsilon$
$F \rightarrow (\ E\ ) \mid \underline{id}$

- FIRST (E) =
- FIRST (E’) =
- FIRST (T) =
- FIRST (T’) =
- FIRST (F) =

First - Example

- $P \rightarrow i \mid c \mid n\ T\ S$
- $Q \rightarrow P \mid a\ S \mid b\ S\ c\ S\ T$
- $R \rightarrow b \mid \varepsilon$
- $S \rightarrow c \mid R\ n \mid \varepsilon$
- $T \rightarrow R\ S\ q$

- FIRST(P) =
- FIRST(Q) =
- FIRST(R) =
- FIRST(S) =
- FIRST(T) =

First - Example

- $P \rightarrow i \mid c \mid n\ T\ S$
- $Q \rightarrow P \mid a\ S \mid b\ S\ c\ S\ T$
- $R \rightarrow b \mid \varepsilon$
- $S \rightarrow c \mid R\ n \mid \varepsilon$
- $T \rightarrow R\ S\ q$

- FIRST(P) = {i,c,n}
- FIRST(Q) = {i,c,n,a,b}
- FIRST(R) = {b,ε}
- FIRST(S) = {c,b,n,ε}
- FIRST(T) = {b,c,n,q}

First - Example

- $S \rightarrow a\ S\ e \mid S\ T\ S$
- $T \rightarrow R\ S\ e \mid Q$
- $R \rightarrow r\ S\ r \mid \varepsilon$
- $Q \rightarrow S\ T \mid \varepsilon$

- FIRST(S) =
- FIRST(R) =
- FIRST(T) =
- FIRST(Q) =

First - Example

- $S \rightarrow a\ S\ e \mid S\ T\ S$
- $T \rightarrow R\ S\ e \mid Q$
- $R \rightarrow r\ S\ r \mid \varepsilon$
- $Q \rightarrow S\ T \mid \varepsilon$

- FIRST(S) = {a}
- FIRST(R) = {r, ε}
- FIRST(T) = {r, a, ε}
- FIRST(Q) = {a, ε}

Any Question?