Syntax Analysis

Outline

- Context-Free Grammars (CFGs)
- Parsing
  - Top-Down
    - Recursive Descent
    - Table-Driven
  - Bottom-Up
    - LR Parsing Algorithm
    - How to Build LR Tables
    - Parser Generators
- Grammar Issues for Programming Languages

Top-Down Parsing

- LL Grammars - A subclass of all CFGs
- Recursive-Descent Parsers - Programmed “by hand”
- Non-Recursive Predictive Parsers - Table Driven
- Simple, Easy to Build, Better Error Handling

Bottom-Up Parsing

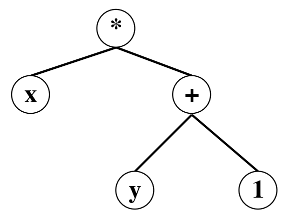
- LR Grammars - A larger subclass of CFGs
- Complex Parsing Algorithms - Table Driven
- Table Construction Techniques
- Parser Generators use this technique
- Faster Parsing, Larger Set of Grammars
- Complex
- Error Reporting is Tricky

Output of Parser?

Succeed is string is recognized
    ... and fail if syntax errors

Syntax Errors?
    Good, descriptive, helpful message!
    Recover and continue parsing!

Build a “Parse Tree” (also called “derivation tree”)



Build Abstract Syntax Tree (AST)
    In memory (with objects, pointers)
    Output to a file

Execute Semantic Actions
    Build AST
    Type Checking
    Generate Code
    Don’t build a tree at all!

Errors in Programs

Lexical

```
if x<1 thenn y = 5:
```

“Typos”

Syntactic

```
if ((x<1) & (y>5))) ...
{ ... { ... _ ... }
```

Semantic

```
if (x+5) then ...
```

Type Errors
Undefined IDs, etc.

Logical Errors

```
if (i<9) then ...
```

Should be <= not <
Bugs
Compiler cannot detect Logical Errors

Compiler

Always halts
Any checks guaranteed to terminate
"Decidable"

Other Program Checking Techniques

Debugging
Testing
Correctness Proofs

"Partially Decidable"

Okay? $\Rightarrow$ The test terminates.
Not Okay? $\Rightarrow$ The test may not terminate!
You may need to run some programs to see if they are okay.

Requirements

Detect All Errors (Except Logical!)

Messages should be helpful.

Difficult to produce clear messages!

Example:

```
Syntax Error
```

Example:

```
Line 23: Unmatched Paren
    if ((x == 1) then
       ^
```

Compiler Should Recover

Keep going to find more errors

Example:

```
x := (a + 5)) * (b + 7));
```

This error missed

Error detected here

We're in the middle of a statement
Skip tokens until we see a ";"
Resume Parsing
Misses a second error... Oh, well...
Checks most of the source

Difficult to generate clear and accurate error messages.

Example

```
function foo () {
  ...
  if (...) {
    ...
  } else {
    ...
                 Missing } here
  ...
}
<eof>            Not detected until here
```

Example

```
var myVarr: Integer;     Misspelled ID here
...
x := myVar;              Detected here as "Undeclared ID"
...
```


For Mature Languages

Catalog common errors
Statistical studies
Tailor compiler to handle common errors well

Statement *terminators* versus *separators*

Terminators: C, Java, PCAT **{A;B;C;}**
Separators: Pascal, Smalltalk, Haskell

Pascal Examples

```
begin
    var t: Integer;
    t := x;
    x := y;
    y := t          Tend to insert a ; here
end
```

```
if (...) then
    x := 1          Tend to insert a ; here
else
    y := 2;
z := 3;
```

```
function foo (x: Integer; y: Integer)...
                          Tend to put a comma here
```

Error-Correcting Compilers

- Issue an error message
- Fix the problem
- Produce an executable

Example

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

Is this a good idea???

Compiler *guesses* the programmer's intent
A shifting notion of what constitutes a correct / legal / valid program
May encourage programmers to get sloppy
Declarations provide redundancy
⇒ Increased reliability

Error Avalanche

One error generates a cascade of messages

Example

```
x := 5 while ( a == b ) do
       ^
       Expecting ;
             ^
             Expecting ;
               ^
               Expecting ;
```

The real messages may be buried under the avalanche.
Missing **`#include`** or **`import`** will also cause an avalanche.

Approaches:

Only print 1 message per token [or per line of source]
Only print a particular message once

```
Error: Variable "myVarr" is undeclared
       All future notices for this ID have been suppressed
```

Abort the compiler after 50 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

- Simple to implement
- Commonly used
- The key...
  - Good set of synchronizing tokens
  - Knowing what to do then
- May skip over large sections of source

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
    - ...constantly inserting tokens
    - ...and never scanning the actual source

Error Recovery Approaches: Error Productions

Augment the CFG with "Error Productions"

Now the CFG accepts anything!

If "error productions" are used...
  Their actions:
```
{ print ("Error...") }
```

Used with...
- LR (Bottom-up) parsing
- Parser Generators

Error Recovery Approaches: Global Correction

Theoretical Approach

Find the minimum change to the source to yield a valid program
  (Insert tokens, delete tokens, swap adjacent tokens)

Impractical algorithms - too time consuming

CFG: Context Free Grammars

Example Rule:
  Stmt → **if** Expr **then** Stmt **else** Stmt

Terminals
  Keywords
    else "else"
  Token Classes
    ID INTEGER REAL
  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 Alternatives

E → E + E
E → (E)
E → - E
E → ID

E → E + E
→ (E)
→ - E
→ ID

E → E + E
| (E)
| - E
| ID

E → E + E | (E) | - E | ID

All Notations are Equivalent

Notational Conventions

Terminals
a b c ...

Nonterminals
A B C ...
S
Expr

Grammar Symbols (Terminals or Nonterminals)
X Y Z U V W ...

Strings of Symbols
α β γ ...

A sequence of zero Or more terminals And nonterminals

Strings of Terminals
x y z u v w ...

Including ε

Examples
A → α B
A rule whose righthand side ends with a nonterminal
A → x α
A rule whose righthand side begins with a string of terminals (call it "x")

Derivations

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

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

E ⇒ (E) ⇒ (E*E) ⇒ (id*E) ⇒ (id*id)

"Sentential Forms"

A sequence of terminals and nonterminals in a derivation
(id*E)

Derives in one step ⇒

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

$$\alpha A \gamma \Rightarrow \alpha \beta \gamma$$

Any sentential form containing a nonterminal (call it A)
... such that A matches the nonterminal in some rule.

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

$$E \Rightarrow^* (id*id)$$

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."

$$G_1 \approx G_2 \text{ whenever } L(G_1) = L(G_2)$$

In making a derivation...

Choose which nonterminal to expand

Choose which rule to apply

Leftmost Derivations

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

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

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

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

```
     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

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."

Bottom-Up Parsing

Bottom-up parsers discover rightmost derivations!

Parser moves from input string back to S.

Follow $S \Rightarrow_{RM}{}^{*} w$ in reverse.

At each step in a rightmost derivation, we have

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

String of terminals (i.e., the rest of the input, which we have not yet seen)

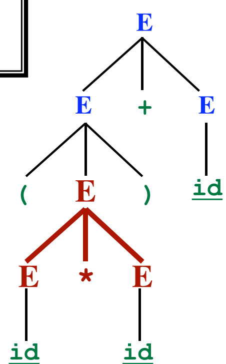
Parse Trees

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



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

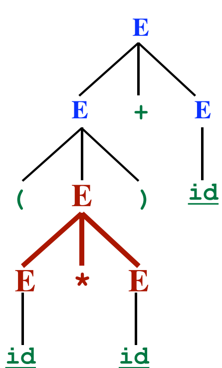
Parse Trees

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 Trees

Two choices at each step in a derivation...
- Which non-terminal to expand
- Which rule to use in replacing it

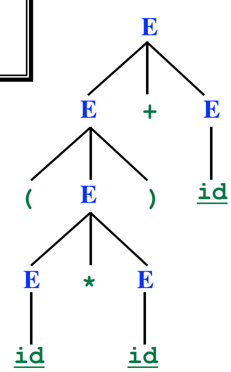
The parse tree remembers only this

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

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

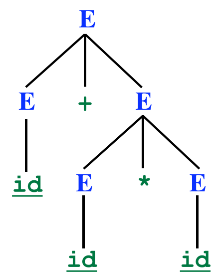
Ambuiguous Grammars

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

Input: `id+id*id`

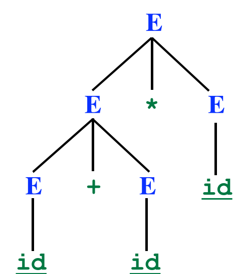
Leftmost Derivation #1

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.

Translating a Regular Expression into a CFG

First build the NFA.

For every state in the NFA...
Make a nonterminal in the grammar

For every edge labeled `c` from A to B...
Add the rule
$A \rightarrow$ `c`B

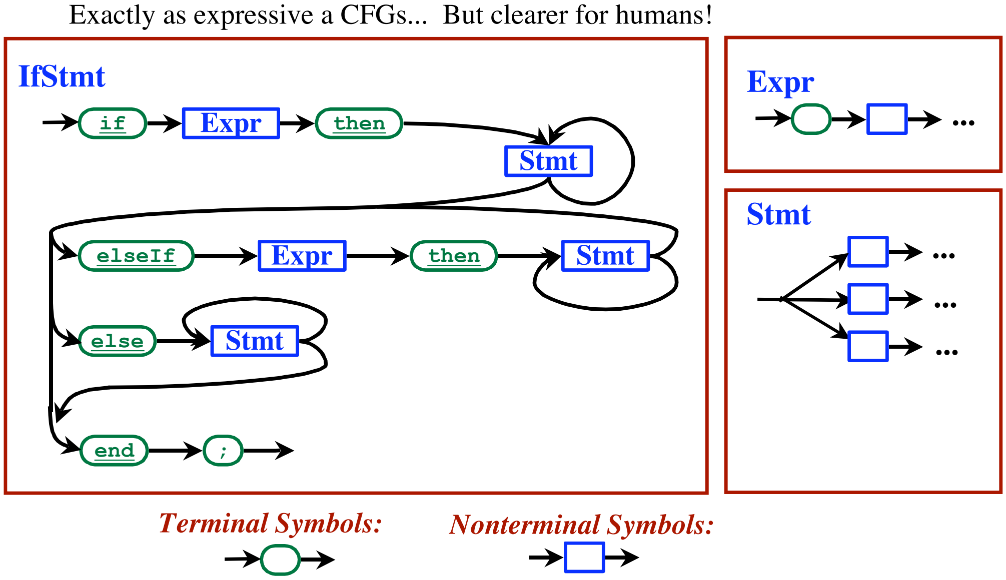
For every edge labeled ε from A to B...
Add the rule
$A \rightarrow B$

For every final state B...
Add the rule
$B \rightarrow \varepsilon$

Recursive Transition Networks

Regular Expressions ⇔ NFA ⇔ DFA

Context-Free Grammar ⇔ Recursive Transition Networks
Exactly as expressive a CFGs... But clearer for humans!



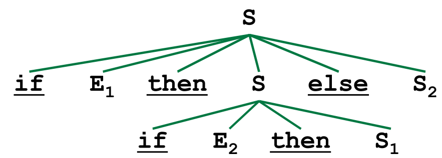
The Dangling “Else” Problem

This grammar is ambiguous!

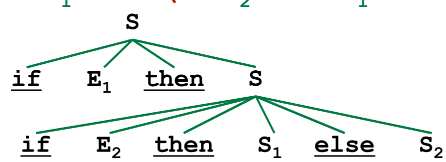
Stmt → **if** Expr **then** Stmt
→ **if** Expr **then** Stmt **else** Stmt
→ ...Other Stmt Forms...

Example String: `if E1 then if E2 then S1 else S2`

Interpretation #1: `if E1 then (if E2 then S1) else S2`



Interpretation #2: `if E1 then (if E2 then S1 else S2)`



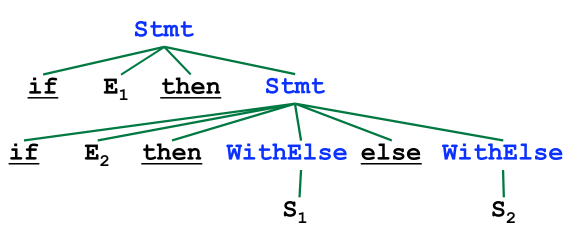
The Dangling “Else” Problem

Goal: “Match **else**-clause to the closest **if** without an **else**-clause already.”
Solution:

Stmt → **if** Expr **then** Stmt
→ **if** Expr **then** WithElse **else** Stmt
→ ...Other Stmt Forms...
WithElse → **if** Expr **then** WithElse **else** WithElse
→ ...Other Stmt Forms...

Any Stmt occurring between **then** and **else** must have an **else**.
i.e., the Stmt must not end with “**then** Stmt”.

Interpretation #2: `if E1 then (if E2 then S1 else S2)`



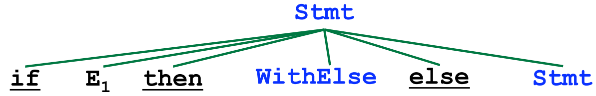
The Dangling "Else" Problem

Goal: "Match `else`-clause to the closest `if` without an `else`-clause already."
Solution:

Stmt	→	`if` Expr `then` Stmt
	→	`if` Expr `then` WithElse `else` Stmt
	→	...Other Stmt Forms...
WithElse	→	`if` Expr `then` WithElse `else` WithElse
	→	...Other Stmt Forms...

Any Stmt occurring between `then` and `else` must have an `else`.
i.e., the Stmt must not end with "`then` Stmt".

Interpretation #1: `if` E_1 `then` (`if` E_2 `then` S_1) `else` S_2



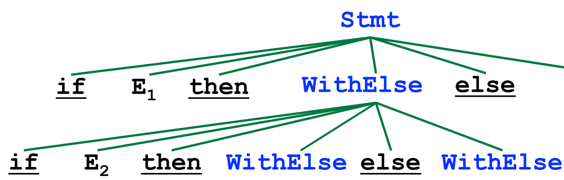
The Dangling "Else" Problem

Goal: "Match `else`-clause to the closest `if` without an `else`-clause already."
Solution:

Stmt	→	`if` Expr `then` Stmt
	→	`if` Expr `then` WithElse `else` Stmt
	→	...Other Stmt Forms...
WithElse	→	`if` Expr `then` WithElse `else` WithElse
	→	...Other Stmt Forms...

Any Stmt occurring between `then` and `else` must have an `else`.
i.e., the Stmt must not end with "`then` Stmt".

Interpretation #1: `if` E_1 `then` (`if` E_2 `then` S_1) `else` S_2



The Dangling "Else" Problem

Goal: "Match `else`-clause to the closest `if` without an `else`-clause already."

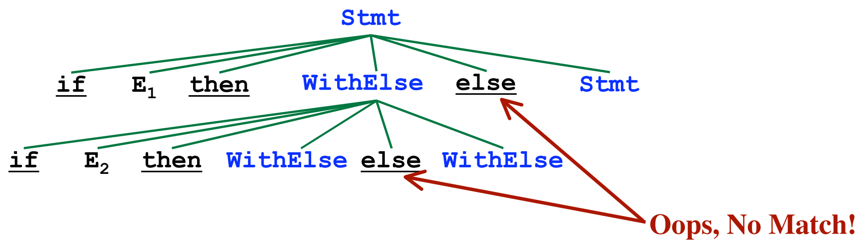
Solution:

Stmt	→	`if` Expr `then` Stmt
	→	`if` Expr `then` WithElse `else` Stmt
	→	...Other Stmt Forms...
WithElse	→	`if` Expr `then` WithElse `else` WithElse
	→	...Other Stmt Forms...

Any Stmt occurring between `then` and `else` must have an `else`.

i.e., the Stmt must not end with "`then` Stmt".

Interpretation #1: `if` E_1 `then (if` E_2 `then` S_1`) else` S_2



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!

Option 1:

"Backtracking"

Made a bad decision

Back up and try another choice

Option 2:

Always make the right choice.

Never have to backtrack: "Predictive Parser"

Possible for some grammars (LL Grammars)

May be able to fix some grammars (but not others)

- Eliminate Left Recursion
- Left-Factor the Rules

Backtracking

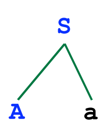
Input: **aabbde**

S

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

Backtracking

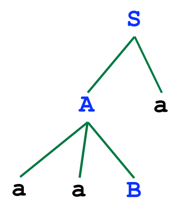
Input: **aabbde**



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

Backtracking

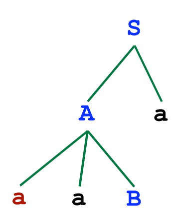
Input: **`aabbde`**



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

Backtracking

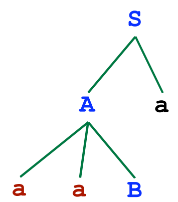
Input: **`aabbde`**



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

Backtracking

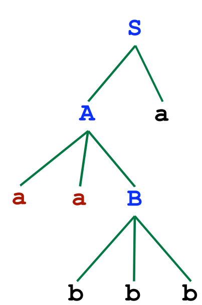
Input: **aabbde**



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

Backtracking

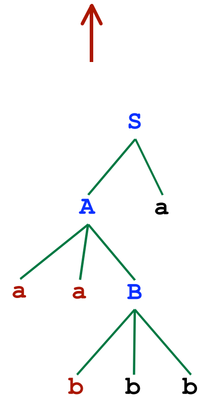
Input: **aabbde**



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

Backtracking

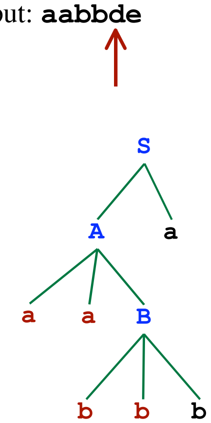
Input: **aabbde**



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

Backtracking

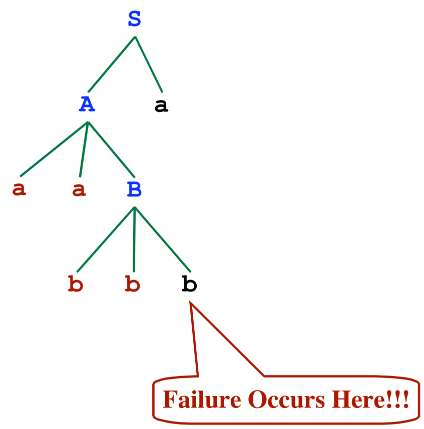
Input: **aabbde**



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

Backtracking

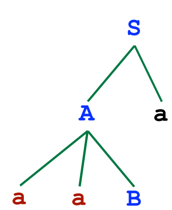
Input: **aabbde**



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

Backtracking

Input: **aabbde**

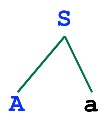


1. S → Aa
2. → Ce
3. A → aaB
4. → aaba
5. B → bbb
6. C → aaD
7. D → bbd

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

Backtracking

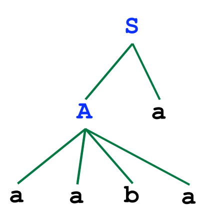
Input: `aabbde`



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

Backtracking

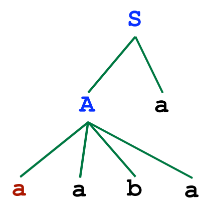
Input: `aabbde`



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

Backtracking

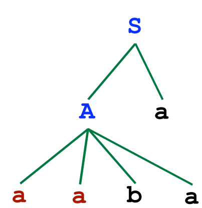
Input: **`aabbde`**



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

Backtracking

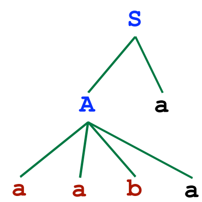
Input: **`aabbde`**



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

Backtracking

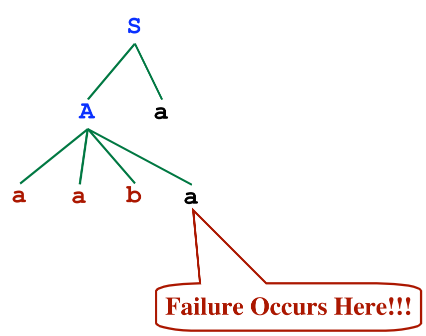
Input: **aabbde**



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

Backtracking

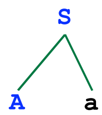
Input: **aabbde**



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

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**

Backtracking

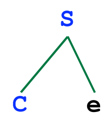
Input: **aabbde**

S

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

Backtracking

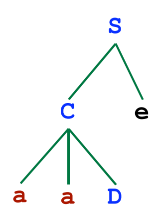
Input: **aabbde**



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

Backtracking

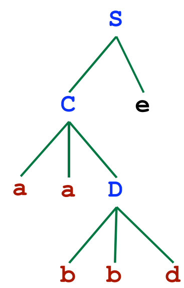
Input: **aabbde**



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

Backtracking

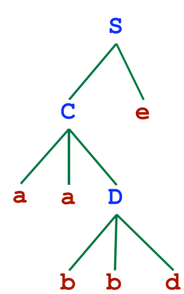
Input: **aabbde**



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

Backtracking

Input: **aabbde**



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

Predictive Parsing

Will never backtrack!

Requirement:

For every rule:

$$A \rightarrow \alpha_1 \mid \alpha_2 \mid \alpha_3 \mid \ldots \mid \alpha_N$$

We must be able to choose the correct alternative
by looking only at the next symbol

May peek ahead to the next symbol (token).

Example

```
A   → aB
    → cD
    → E
```

Assuming **a**,**c** $\notin$ FIRST (E)

Example

```
Stmt → if Expr ...
     → for LValue ...
     → while Expr ...
     → return Expr ...
     → ID ...
```


Predictive Parsing

LL(1) Grammars

Can do predictive parsing

Can select the right rule

Looking at only the next 1 input symbol

Predictive Parsing

LL(1) Grammars

Can do predictive parsing

Can select the right rule

Looking at only the next 1 input symbol

LL(k) Grammars

Can do predictive parsing

Can select the right rule

Looking at only the next k input symbols

Syntax Analysis - Part 1

Predictive Parsing

LL(1) Grammars

Can do predictive parsing

Can select the right rule

Looking at only the next 1 input symbol

LL(k) Grammars

Can do predictive parsing

Can select the right rule

Looking at only the next k input symbols

Techniques to modify the grammar:

- Left Factoring
- Removal of Left Recursion

Predictive Parsing

LL(1) Grammars

Can do predictive parsing
Can select the right rule
Looking at only the next 1 input symbol

LL(k) Grammars

Can do predictive parsing
Can select the right rule
Looking at only the next k input symbols

Techniques to modify the grammar:

- Left Factoring
- Removal of Left Recursion

But these may not be enough!

Predictive Parsing

LL(1) Grammars

Can do predictive parsing
Can select the right rule
Looking at only the next 1 input symbol

LL(k) Grammars

Can do predictive parsing
Can select the right rule
Looking at only the next k input symbols

Techniques to modify the grammar:

- Left Factoring
- Removal of Left Recursion

But these may not be enough!

LL(k) Language

Can be described with an LL(k) grammar.

Left-Factoring

Problem:

Stmt → **if** Expr **then** Stmt **else** Stmt

→ **if** Expr **then** Stmt

→ OtherStmt

With predictive parsing, we need to know which rule to use!
(While looking at just the next token)

Left-Factoring

Problem:

Stmt → **if** Expr **then** Stmt **else** Stmt

→ **if** Expr **then** Stmt

→ OtherStmt

With predictive parsing, we need to know which rule to use!
(While looking at just the next token)

Solution:

Stmt → **if** Expr **then** Stmt ElsePart

→ OtherStmt

ElsePart → **else** Stmt | ε

Left-Factoring

Problem:

Stmt → **if** Expr **then** Stmt **else** Stmt

→ **if** Expr **then** Stmt

→ OtherStmt

With predictive parsing, we need to know which rule to use!
(While looking at just the next token)

Solution:

Stmt → **if** Expr **then** Stmt ElsePart

→ OtherStmt

ElsePart → **else** Stmt | ε

General Approach:

Before: A $\rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \alpha\beta_3 \mid \ldots \mid \delta_1 \mid \delta_2 \mid \delta_3 \mid \ldots$

After: A $\rightarrow \alpha C \mid \delta_1 \mid \delta_2 \mid \delta_3 \mid \ldots$
C $\rightarrow \beta_1 \mid \beta_2 \mid \beta_3 \mid \ldots$

Left-Factoring

Problem:

$\underbrace{\text{Stmt}}_{A} \rightarrow \underbrace{\textbf{if}\ \text{Expr}\ \textbf{then}\ \text{Stmt}}_{\alpha}\ \underbrace{\textbf{else}\ \text{Stmt}}_{\beta_1}$

$\rightarrow \underbrace{\textbf{if}\ \text{Expr}\ \textbf{then}\ \text{Stmt}}_{\alpha}\ \underbrace{\varepsilon}_{\beta_2}$

$\rightarrow \underbrace{\text{OtherStmt}}_{\delta_1}$

With predictive parsing, we need to know which rule to use!
(While looking at just the next token)

Solution:

Stmt → **if** Expr **then** Stmt ElsePart

→ OtherStmt

ElsePart → **else** Stmt | ε

General Approach:

Before: A $\rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \alpha\beta_3 \mid \ldots \mid \delta_1 \mid \delta_2 \mid \delta_3 \mid \ldots$

After: A $\rightarrow \alpha C \mid \delta_1 \mid \delta_2 \mid \delta_3 \mid \ldots$
C $\rightarrow \beta_1 \mid \beta_2 \mid \beta_3 \mid \ldots$

Left-Factoring

Problem:

$\underbrace{\text{Stmt}}_{A} \to \underbrace{\textbf{if}\ \text{Expr}\ \textbf{then}\ \text{Stmt}}_{\alpha}\ \underbrace{\textbf{else}\ \text{Stmt}}_{\beta_1}$

$\to \underbrace{\textbf{if}\ \text{Expr}\ \textbf{then}\ \text{Stmt}}_{\alpha}\ \underbrace{\varepsilon}_{\beta_2}$

$\to \underbrace{\text{OtherStmt}}_{\delta_1}$

With predictive parsing, we need to know which rule to use!
(While looking at just the next token)

Solution:

$\underbrace{\text{Stmt}}_{A} \to \underbrace{\textbf{if}\ \text{Expr}\ \textbf{then}\ \text{Stmt}}_{\alpha}\ \underbrace{\text{ElsePart}}_{C}$

$\to \underbrace{\text{OtherStmt}}_{\delta_1}$

$\underbrace{\text{ElsePart}}_{C} \to \underbrace{\textbf{else}\ \text{Stmt}}_{\beta_1} \mid \underbrace{\varepsilon}_{\beta_2}$

General Approach:

Before: $A \to \alpha\beta_1 \mid \alpha\beta_2 \mid \alpha\beta_3 \mid \ldots \mid \delta_1 \mid \delta_2 \mid \delta_3 \mid \ldots$

After: $A \to \alpha C \mid \delta_1 \mid \delta_2 \mid \delta_3 \mid \ldots$
$C \to \beta_1 \mid \beta_2 \mid \beta_3 \mid \ldots$

Left Recursion

Whenever

$A \Rightarrow^{+} A\alpha$

Simplest Case: Immediate Left Recursion

Given:

$A \to A\alpha \mid \beta$

Transform into:

$A \to \beta A'$

$A' \to \alpha A' \mid \varepsilon$ where A' is a new nonterminal

More General (but still immediate):

$A \to A\alpha_1 \mid A\alpha_2 \mid A\alpha_3 \mid \ldots \mid \beta_1 \mid \beta_2 \mid \beta_3 \mid \ldots$

Transform into:

$A \to \beta_1 A' \mid \beta_2 A' \mid \beta_3 A' \mid \ldots$

$A' \to \alpha_1 A' \mid \alpha_2 A' \mid \alpha_3 A' \mid \ldots \mid \varepsilon$

Left Recursion in More Than One Step

Example:

S → A<u>f</u> | <u>b</u>
A → A<u>c</u> | S<u>d</u> | <u>e</u>

Is A left recursive? Yes.

Syntax Analysis - Part 1

Left Recursion in More Than One Step

Example:

S → A<u>f</u> | <u>b</u>
A → A<u>c</u> | S<u>d</u> | <u>e</u>

Is A left recursive? Yes.
Is S left recursive?

Left Recursion in More Than One Step

Example:

S → Af | b

A → Ac | Sd | e

Is A left recursive? Yes.

Is S left recursive? Yes, but not immediate left recursion.   S ⇒ Af ⇒ Sdf

Left Recursion in More Than One Step

Example:

S → Af | b

A → Ac | Sd | e

Is A left recursive? Yes.

Is S left recursive? Yes, but not immediate left recursion.   S ⇒ Af ⇒ Sdf

Approach:

Look at the rules for S only (ignoring other rules)... No left recursion.

Left Recursion in More Than One Step

Example:

S → Af | b

A → Ac | Sd | e

Is A left recursive? Yes.

Is S left recursive? Yes, but not immediate left recursion. S ⇒ Af ⇒ Sdf

Approach:

Look at the rules for S only (ignoring other rules)... No left recursion.

Look at the rules for A...

Syntax Analysis - Part 1

Left Recursion in More Than One Step

Example:

S → Af | b

A → Ac | Sd | e

Is A left recursive? Yes.

Is S left recursive? Yes, but not immediate left recursion. S ⇒ Af ⇒ Sdf

Approach:

Look at the rules for S only (ignoring other rules)... No left recursion.

Look at the rules for A...

Do any of A's rules start with S? Yes.

A → Sd

Left Recursion in More Than One Step

Example:

S → Af | b

A → Ac | Sd | e

Is A left recursive? Yes.

Is S left recursive? Yes, but not immediate left recursion. S ⇒ Af ⇒ Sdf

Approach:

Look at the rules for S only (ignoring other rules)... No left recursion.

Look at the rules for A...

Do any of A's rules start with S? Yes.

A → Sd

Get rid of the S. Substitute in the righthand sides of S.

A → Afd | bd

Left Recursion in More Than One Step

Example:

S → Af | b

A → Ac | Sd | e

Is A left recursive? Yes.

Is S left recursive? Yes, but not immediate left recursion. S ⇒ Af ⇒ Sdf

Approach:

Look at the rules for S only (ignoring other rules)... No left recursion.

Look at the rules for A...

Do any of A's rules start with S? Yes.

A → Sd

Get rid of the S. Substitute in the righthand sides of S.

A → Afd | bd

The modified grammar:

S → Af | b

A → Ac | Afd | bd | e

Left Recursion in More Than One Step

Example:

$S \rightarrow A\underline{f} \mid \underline{b}$

$A \rightarrow A\underline{c} \mid S\underline{d} \mid \underline{e}$

Is A left recursive? Yes.

Is S left recursive? Yes, but not immediate left recursion. $S \Rightarrow A\underline{f} \Rightarrow S\underline{df}$

Approach:

Look at the rules for S only (ignoring other rules)... No left recursion.

Look at the rules for A...

Do any of A's rules start with S? Yes.

$A \rightarrow S\underline{d}$

Get rid of the S. Substitute in the righthand sides of S.

$A \rightarrow A\underline{fd} \mid \underline{bd}$

The modified grammar:

$S \rightarrow A\underline{f} \mid \underline{b}$

$A \rightarrow A\underline{c} \mid A\underline{fd} \mid \underline{bd} \mid \underline{e}$

Now eliminate immediate left recursion involving A.

$S \rightarrow A\underline{f} \mid \underline{b}$

$A \rightarrow \underline{bd}A' \mid \underline{e}A'$

$A' \rightarrow \underline{c}A' \mid \underline{fd}A' \mid \underline{\varepsilon}$

Left Recursion in More Than One Step

The Original Grammar:

$S \rightarrow A\underline{f} \mid \underline{b}$

$A \rightarrow A\underline{c} \mid S\underline{d} \mid \underline{e}$

Left Recursion in More Than One Step

The Original Grammar:

S → A**f** | **b**
A → A**c** | S**d** | B**e**
B → A**g** | S**h** | **k**

Assume there are still more nonterminals;
Look at the next one...

Left Recursion in More Than One Step

The Original Grammar:

S → A**f** | **b**
A → A**c** | S**d** | B**e**
B → A**g** | S**h** | **k**

So Far:

S → A**f** | **b**
A → **bd**A' | B**e**A'
A' → **c**A' | **fd**A' | ε

Left Recursion in More Than One Step

The Original Grammar:

S → Af | b
A → Ac | Sd | Be
B → Ag | Sh | k

So Far:

S → Af | b
A → bdA' | BeA'
A' → cA' | fdA' | ε
B → Ag | Sh | k

Look at the B rules next;
Does any righthand side
start with "S"?

Left Recursion in More Than One Step

The Original Grammar:

S → Af | b
A → Ac | Sd | Be
B → Ag | Sh | k

So Far:

S → Af | b
A → bdA' | BeA'
A' → cA' | fdA' | ε
B → Ag | Afh | bh | k

Substitute, using the rules for "S"
Af... | b...

Left Recursion in More Than One Step

The Original Grammar:

S → Af | b
A → Ac | Sd | Be
B → Ag | Sh | k

So Far:

S → Af | b
A → bdA' | BeA'
A' → cA' | fdA' | ε
B → Ag | Afh | bh | k

Does any righthand side start with "A"?

Syntax Analysis - Part 1

Left Recursion in More Than One Step

The Original Grammar:

S → Af | b
A → Ac | Sd | Be
B → Ag | Sh | k

So Far:

S → Af | b
A → bdA' | BeA'
A' → cA' | fdA' | ε
B → Ag | Afh | bh | k

Do this one first.

Left Recursion in More Than One Step

The Original Grammar:

S → Af | b
A → Ac | Sd | Be
B → Ag | Sh | k

So Far:

S → Af | b
A → bdA' | BeA'
A' → cA' | fdA' | ε
B → bdA'g | BeA'g | Afh | bh | k

Substitute, using the rules for "A"
bdA'... | BeA'...

Left Recursion in More Than One Step

The Original Grammar:

S → Af | b
A → Ac | Sd | Be
B → Ag | Sh | k

So Far:

S → Af | b
A → bdA' | BeA'
A' → cA' | fdA' | ε
B → bdA'g | BeA'g | Afh | bh | k

Do this one next.

Left Recursion in More Than One Step

The Original Grammar:

S → Af | b
A → Ac | Sd | Be
B → Ag | Sh | k

So Far:

S → Af | b
A → bdA' | BeA'
A' → cA' | fdA' | ε
B → bdA'g | BeA'g | bdA'fh | BeA'fh | bh | k

Substitute, using the rules for "A"
bdA'... | BeA'...

Left Recursion in More Than One Step

The Original Grammar:

S → Af | b
A → Ac | Sd | Be
B → Ag | Sh | k

Finally, eliminate any immediate Left recursion involving "B"

So Far:

S → Af | b
A → bdA' | BeA'
A' → cA' | fdA' | ε
B → bdA'g | BeA'g | bdA'fh | BeA'fh | bh | k

Left Recursion in More Than One Step

The Original Grammar:

S → Af | b
A → Ac | Sd | Be
B → Ag | Sh | k

So Far:

S → Af | b
A → bdA' | BeA'
A' → cA' | fdA' | ε
B → bdA'gB' | bdA'fhB' | bhB' | kB'
B' → eA'gB' | eA'fhB' | ε

Finally, eliminate any immediate Left recursion involving "B"

Left Recursion in More Than One Step

The Original Grammar:

S → Af | b
A → Ac | Sd | Be | C
B → Ag | Sh | k
C → BkmA | AS | j

If there is another nonterminal, then do it next.

So Far:

S → Af | b
A → bdA' | BeA' | CA'
A' → cA' | fdA' | ε
B → bdA'gB' | bdA'fhB' | bhB' | kB' | CA'gB' | CA'fhB'
B' → eA'gB' | eA'fhB' | ε

Algorithm to Eliminate Left Recursion

Assume the nonterminals are ordered $A_1, A_2, A_3, \ldots$

(In the example: S, A, B)

<u>for</u> <u>each</u> nonterminal A_i **(for i = 1 to N) <u>do</u>**
  <u>for</u> <u>each</u> nonterminal A_j **(for j = 1 to i-1) <u>do</u>**
    Let $A_j \rightarrow \beta_1 \mid \beta_2 \mid \beta_3 \mid \ldots \mid \beta_N$ **be all the rules for** A_j
    <u>if</u> there is a rule of the form
      $A_i \rightarrow A_j\alpha$
    <u>then</u> replace it by
      $A_i \rightarrow \beta_1\alpha \mid \beta_2\alpha \mid \beta_3\alpha \mid \ldots \mid \beta_N\alpha$
    <u>endIf</u>
  <u>endFor</u>
  Eliminate immediate left recursion among the A_i **rules**
<u>endFor</u>

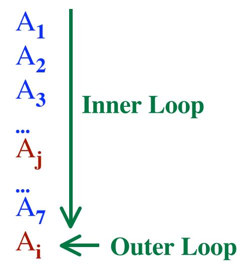


Table-Driven Predictive Parsing Algorithm

Assume that the grammar is LL(1)

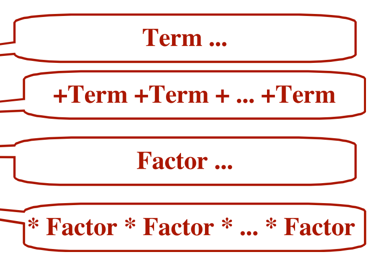
i.e., Backtracking will never be needed

Always know which righthand side to choose (with one look-ahead)

- No Left Recursion
- Grammar is Left-Factored.

Example

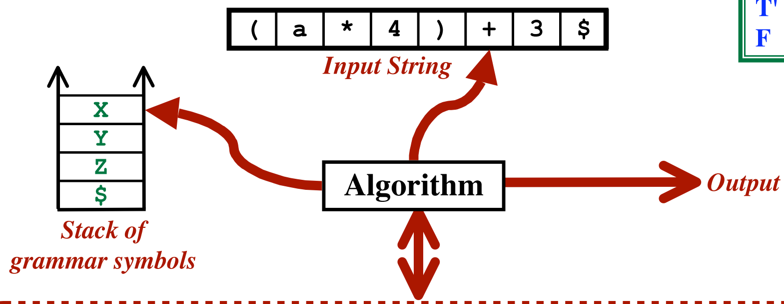
$E \rightarrow T\ E'$
$E' \rightarrow +\ T\ E' \mid \varepsilon$
$T \rightarrow F\ T'$
$T' \rightarrow *\ F\ T' \mid \varepsilon$
$F \rightarrow (\ E\ ) \mid \underline{id}$



Step 1: From grammar, construct table.
Step 2: Use table to parse strings.

E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

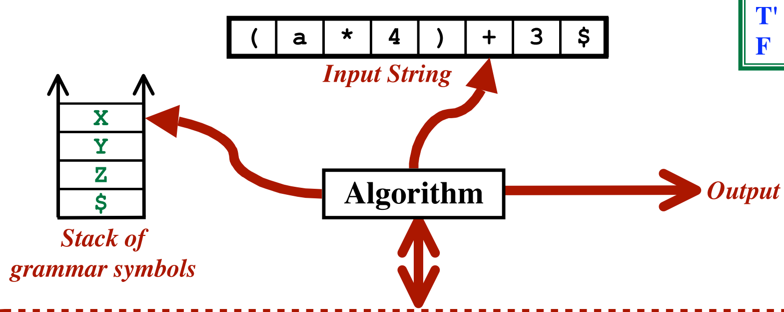
Table-Driven Predictive Parsing Algorithm



Pre-Computed Table:

E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

Table-Driven Predictive Parsing Algorithm



Pre-Computed Table:

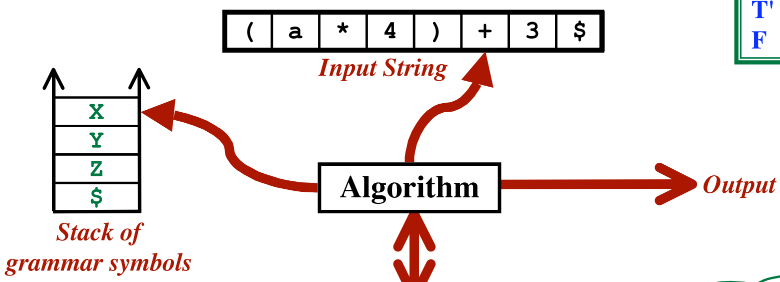
Input Symbols, plus $

Nonterminals

	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)		

E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

Table-Driven Predictive Parsing Algorithm



Pre-Computed Table:

Input Symbols, plus $

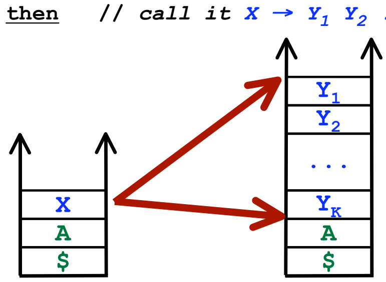
Blank entries indicate ERROR

Nonterminals

	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)		

Predictive Parsing Algorithm

```
Set input ptr to first symbol; Place $ after last input symbol
Push $
Push S
repeat
  X = stack top
  a = current input symbol
  if X is a terminal or X = $ then
    if X == a then
      Pop stack
      Advance input ptr
    else
      Error
    endIf
  elseIf Table[X,a] contains a rule then   // call it X → Y1 Y2 ... YK
    Pop stack
    Push YK
    ...
    Push Y2
    Push Y1
    Print ("X → Y1 Y2 ... YK")
  else    // Table[X,a] is blank
    Syntax Error
  endIf
until X == $
```



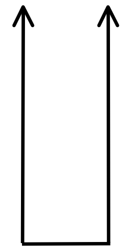
Example

E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | <u>id</u>

Input:

`(id*id)+id`

Output:



(	id	*	id	)	+	id	

	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)		

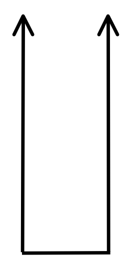
Example

E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | <u>id</u>

Input:

`(id*id)+id`

Output:



(	id	*	id	)	+	id	

Add $ to end of input
Push $
Push E

	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)		

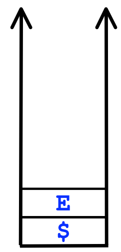
Example

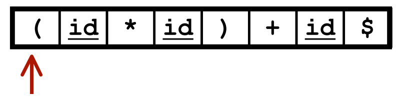
E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

Input:

`(id*id)+id`

Output:





Add $ to end of input
Push $
Push E

	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)		

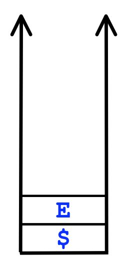
Example

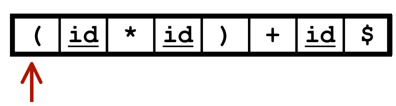
E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

Input:

`(id*id)+id`

Output:





Look at Table [E, '(']
Use rule E→TE'
Pop E
Push E'
Push T
Print E→TE'

	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)		

E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

Example

Input:

`(id*id)+id`

Output:

E → T E'

(	id	*	id	)	+	id	$

Stack:

T
E'
$

Look at Table [E, '(']
Use rule E→TE'
Pop E
Push E'
Push T
Print E→TE'

	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)		

E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

Example

Input:

`(id*id)+id`

Output:

E → T E'

(	id	*	id	)	+	id	$

Stack:

T
E'
$

Table [T, '('] = T→FT'
Pop T
Push T'
Push F
Print T→FT'

	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)		

Example

E	→ T E'
E'	→ + T E' | ε
T	→ F T'
T'	→ * F T' | ε
F	→ (E) | id

Input:

(id*id)+id

Output:

E → T E'
T → F T'

Stack: F, T', E', $

(	id	*	id	)	+	id	$

Table [T, '('] = T→FT'
Pop T
Push T'
Push F
Print T→FT'

	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)		

Example

E	→ T E'
E'	→ + T E' | ε
T	→ F T'
T'	→ * F T' | ε
F	→ (E) | id

Input:

(id*id)+id

Output:

E → T E'
T → F T'

Stack: F, T', E', $

(	id	*	id	)	+	id	$

Table [F, '('] = F→(E)
Pop F
Push (
Push E
Push)
Print F→(E)

	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)		

Example

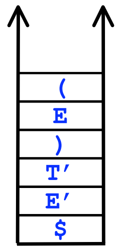
Input:

(id*id)+id

Output:

E → T E'
T → F T'
F → (E)

E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id



(	id	*	id	)	+	id	$

Table [F, '('] = F→(E)
Pop F
Push)
Push E
Push (
Print F→(E)

	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)		

Example

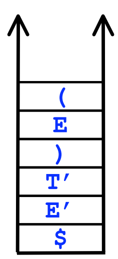
Input:

(id*id)+id

Output:

E → T E'
T → F T'
F → (E)

E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id



(	id	*	id	)	+	id	$

Top of Stack matches next input
Pop and Scan

	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)		

Example

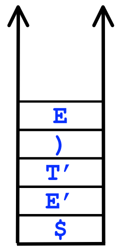
E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

Input:

(id*id)+id

Output:

E → T E'
T → F T'
F → (E)



(	id	*	id	)	+	id	$	

Top of Stack matches next input
Pop and Scan

	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)		

Example

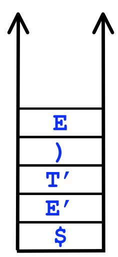
E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

Input:

(id*id)+id

Output:

E → T E'
T → F T'
F → (E)



(	id	*	id	)	+	id	$	

Table [E, id] = E→TE'
Pop E
Push E'
Push T
Print E→TE'

	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)		

E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

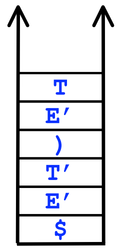
Input:

`(id*id)+id`

Output:

E → T E'
T → F T'
F → (E)
E → T E'

Example



(	id	*	id	)	+	id	$

Table [E, id] = E→TE'
Pop E
Push E'
Push T
Print E→TE'

	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)		

E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

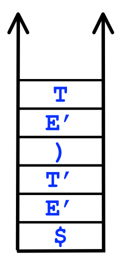
Input:

`(id*id)+id`

Output:

E → T E'
T → F T'
F → (E)
E → T E'

Example



(	id	*	id	)	+	id	$

Table [T, id] = T→FT'
Pop T
Push T'
Push F
Print T→FT'

	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)		

E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

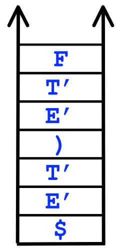
Example

Input:

(id*id)+id

Output:

E → T E'
T → F T'
F → (E)
E → T E'
T → F T'



(	id	*	id	)	+	id	$

Table [T, id] = T→FT'
Pop T
Push T'
Push F
Print T→FT'

	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)		

E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

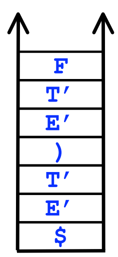
Example

Input:

(id*id)+id

Output:

E → T E'
T → F T'
F → (E)
E → T E'
T → F T'



(	id	*	id	)	+	id	$

Table [F, id] = F→id
Pop F
Push id
Print F→id

	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)		

E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

Example

Input:

(id*id)+id

Output:

E → T E'
T → F T'
F → (E)
E → T E'
T → F T'
F → id

Stack:

id
T'
E'
)
T'
E'
$

(	id	*	id	)	+	id	$

Table [F, id] = F→id
Pop F
Push id
Print F→id

	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)		

E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

Example

Input:

(id*id)+id

Output:

E → T E'
T → F T'
F → (E)
E → T E'
T → F T'
F → id

Stack:

id
T'
E'
)
T'
E'
$

(	id	*	id	)	+	id	$

Top of Stack matches next input
Pop and Scan

	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)		

E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

Example

Input:

```
(id*id)+id
```

Output:

E → T E'
T → F T'
F → (E)
E → T E'
T → F T'
F → id

Stack
T'
E'
)
T'
E'
$

(	id	*	id	)	+	id	$

Top of Stack matches next input
Pop and Scan

	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)		

E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

Example

Input:

```
(id*id)+id
```

Output:

E → T E'
T → F T'
F → (E)
E → T E'
T → F T'
F → id

Stack
T'
E'
)
T'
E'
$

(	id	*	id	)	+	id	$

Table [T', ''] =* T'→*FT'
Pop T'
Push T'
Push F
Push ''*
Print T'→*FT'

	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)		

Example

E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

Input:

(id*id)+id

Output:

E → T E'
T → F T'
F → (E)
E → T E'
T → F T'
F → id
T' → * F T'

Stack:

*
F
T'
E'
)
T'
E'
$

(	id	*	id	)	+	id	$

Table [T', ''] = T'→*FT'*
Pop T'
Push T'
Push F
Push ''*
*Print T'→*FT'*

	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)		

Example

E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

Input:

(id*id)+id

Output:

E → T E'
T → F T'
F → (E)
E → T E'
T → F T'
F → id
T' → * F T'

Stack:

*
F
T'
E'
)
T'
E'
$

(	id	*	id	)	+	id	$

Top of Stack matches next input
Pop and Scan

	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)		

Example

E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

Input:

(id*id)+id

Output:

E → T E'
T → F T'
F → (E)
E → T E'
T → F T'
F → id
T' → * F T'

Stack: F, T', E',), T', E', $

(	id	*	id	)	+	id	$

Top of Stack matches next input
Pop and Scan

	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)		

Example

E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

Input:

(id*id)+id

Output:

E → T E'
T → F T'
F → (E)
E → T E'
T → F T'
F → id
T' → * F T'

Stack: F, T', E',), T', E', $

(	id	*	id	)	+	id	$

Table [F, id] = F→id
Pop F
Push id
Print F→id

	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)		

E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

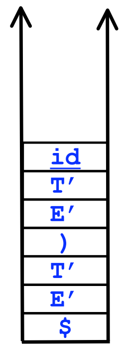
Example

Input:

(id*id)+id

Output:

E → T E'
T → F T'
F → (E)
E → T E'
T → F T'
F → id
T' → * F T'
F → id



| (| id | * | id |) | + | id | $ |

Table [F, id] = F→id
Pop F
Push id
Print F→id

	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)		

E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

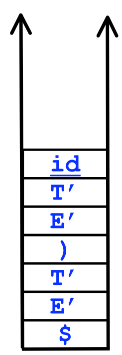
Example

Input:

(id*id)+id

Output:

E → T E'
T → F T'
F → (E)
E → T E'
T → F T'
F → id
T' → * F T'
F → id



| (| id | * | id |) | + | id | $ |

Top of Stack matches next input
Pop and Scan

	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)		

Example

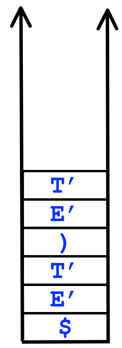
E	→	T E'
E'	→	+ T E' | ε
T	→	F T'
T'	→	* F T' | ε
F	→	(E) | id

Input:

`(id*id)+id`

Output:

E	→	T E'
T	→	F T'
F	→	(E)
E	→	T E'
T	→	F T'
F	→	id
T'	→	* F T'
F	→	id



(	id	*	id	)	+	id	$

Top of Stack matches next input
Pop and Scan

	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)		

Example

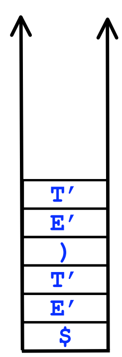
E	→	T E'
E'	→	+ T E' | ε
T	→	F T'
T'	→	* F T' | ε
F	→	(E) | id

Input:

`(id*id)+id`

Output:

E	→	T E'
T	→	F T'
F	→	(E)
E	→	T E'
T	→	F T'
F	→	id
T'	→	* F T'
F	→	id



(	id	*	id	)	+	id	$

Table [T', ')'] = T'→ε
Pop T'
Push <nothing>
Print T'→ε

	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)		

Example

E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

Input:

(id*id)+id

Output:

E → T E'
T → F T'
F → (E)
E → T E'
T → F T'
F → id
T' → * F T'
F → id
T' → ε

Stack:

E'
)
T'
E'
$

(	id	*	id	)	+	id	$

Table [T', ')'] = T'→ε
Pop T'
Push <nothing>
Print T'→ε

	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)		

Example

E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

Input:

(id*id)+id

Output:

E → T E'
T → F T'
F → (E)
E → T E'
T → F T'
F → id
T' → * F T'
F → id
T' → ε

Stack:

E'
)
T'
E'
$

(	id	*	id	)	+	id	$

Table [E', ')'] = E'→ε
Pop E'
Push <nothing>
Print E'→ε

	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)		

E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

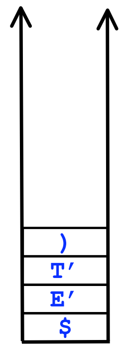
Example

Input:

`(id*id)+id`

Output:

E → T E'
T → F T'
F → (E)
E → T E'
T → F T'
F → id
T' → * F T'
F → id
T' → ε
E' → ε



(	id	*	id	)	+	id	$

Table [E', ')'] = E'→ε
Pop E'
Push <nothing>
Print E'→ε

	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)		

E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

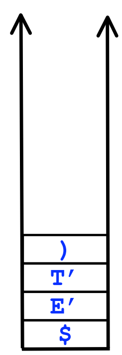
Example

Input:

`(id*id)+id`

Output:

E → T E'
T → F T'
F → (E)
E → T E'
T → F T'
F → id
T' → * F T'
F → id
T' → ε
E' → ε



(	id	*	id	)	+	id	$

Top of Stack matches next input
Pop and Scan

	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)		

Example

E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

Input:

(id*id)+id

Output:

E → T E'
T → F T'
F → (E)
E → T E'
T → F T'
F → id
T' → * F T'
F → id
T' → ε
E' → ε

T'
E'
$

(	id	*	id	)	+	id	$

Top of Stack matches next input
Pop and Scan

	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)		

Example

E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

Input:

(id*id)+id

Output:

E → T E'
T → F T'
F → (E)
E → T E'
T → F T'
F → id
T' → * F T'
F → id
T' → ε
E' → ε

T'
E'
$

(	id	*	id	)	+	id	$

Table [T', '+'] = T'→ε
Pop T'
Push <nothing>
Print T'→ε

	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)		

E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

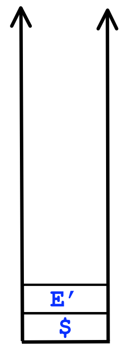
Example

Input:

`(id*id)+id`

Output:

E → T E'
T → F T'
F → (E)
E → T E'
T → F T'
F → id
T' → * F T'
F → id
T' → ε
E' → ε
T' → ε



(	id	*	id	)	+	id	$

Table [T', '+'] = T'→ε
Pop T'
Push <nothing>
Print T'→ε

	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)		

E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

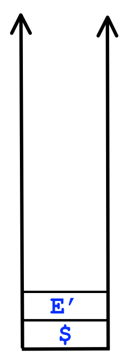
Example

Input:

`(id*id)+id`

Output:

E → T E'
T → F T'
F → (E)
E → T E'
T → F T'
F → id
T' → * F T'
F → id
T' → ε
E' → ε
T' → ε



(	id	*	id	)	+	id	$

Table [E', '+'] = E'→+TE'
Pop E'
Push E'
Push T
Push '+'
Print E'→+TE'

	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)		

Example

E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

Input:

(id*id)+id

Output:

E → T E'
T → F T'
F → (E)
E → T E'
T → F T'
F → id
T' → * F T'
F → id
T' → ε
E' → ε
T' → ε
E' → + T E'

Stack:

+
T
E'
$

(	id	*	id	)	+	id	$

Table [E', '+'] = E'→+TE'
Pop E'
Push E'
Push T
Push '+'
Print E'→+TE'

	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)		

Syntax Analysis - Part 1

Example

E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

Input:

(id*id)+id

Output:

E → T E'
T → F T'
F → (E)
E → T E'
T → F T'
F → id
T' → * F T'
F → id
T' → ε
E' → ε
T' → ε
E' → + T E'

Stack:

+
T
E'
$

(	id	*	id	)	+	id	$

Top of Stack matches next input
Pop and Scan

	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)		

Example

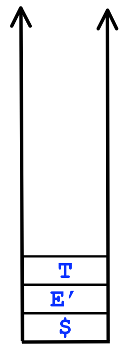
E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

Input:

(id*id)+id

Output:

E → T E'
T → F T'
F → (E)
E → T E'
T → F T'
F → id
T' → * F T'
F → id
T' → ε
E' → ε
T' → ε
E' → + T E'



(	id	*	id	)	+	id	$

Top of Stack matches next input
Pop and Scan

	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)		

Example

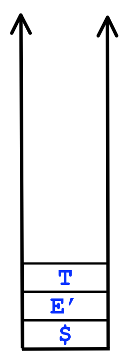
E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

Input:

(id*id)+id

Output:

E → T E'
T → F T'
F → (E)
E → T E'
T → F T'
F → id
T' → * F T'
F → id
T' → ε
E' → ε
T' → ε
E' → + T E'



(	id	*	id	)	+	id	$

Table [T, id] = T→FT'
Pop T
Push T'
Push F
Print T→FT'

	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)		

E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

Example

Input:

(id*id)+id

Output:

E → T E'
T → F T'
F → (E)
E → T E'
T → F T'
F → id
T' → * F T'
F → id
T' → ε
E' → ε
T' → ε
E' → + T E'
T → F T'

Stack (top to bottom): F, T', E', $

(	id	*	id	)	+	id	$

↑ (at the second id)

Table [T, id] = T→FT'
Pop T
Push T'
Push F
Print T→FT'

	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)		

E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

Example

Input:

(id*id)+id

Output:

E → T E'
T → F T'
F → (E)
E → T E'
T → F T'
F → id
T' → * F T'
F → id
T' → ε
E' → ε
T' → ε
E' → + T E'
T → F T'

Stack (top to bottom): F, T', E', $

(	id	*	id	)	+	id	$

↑ (at the second id)

Table [F, id] = F→id
Pop F
Push id
Print F→id

	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)		

Example

E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

Input:

(id*id)+id

Output:

E → T E'
T → F T'
F → (E)
E → T E'
T → F T'
F → id
T' → * F T'
F → id
T' → ε
E' → ε
T' → ε
E' → + T E'
T → F T'
F → id

Stack:

id
T'
E'
$

(	id	*	id	)	+	id	$

Table [F, id] = F→id
Pop F
Push id
Print F→id

	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)		

Example

E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

Input:

(id*id)+id

Output:

E → T E'
T → F T'
F → (E)
E → T E'
T → F T'
F → id
T' → * F T'
F → id
T' → ε
E' → ε
T' → ε
E' → + T E'
T → F T'
F → id

Stack:

id
T'
E'
$

(	id	*	id	)	+	id	$

Top of Stack matches next input
Pop and Scan

	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)		

Example

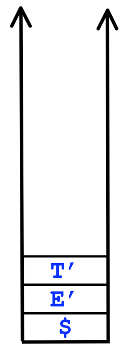
E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

Input:

(id*id)+id

Output:

E → T E'
T → F T'
F → (E)
E → T E'
T → F T'
F → id
T' → * F T'
F → id
T' → ε
E' → ε
T' → ε
E' → + T E'
T → F T'
F → id



(	id	*	id	)	+	id	$

Top of Stack matches next input
Pop and Scan

	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)		

Example

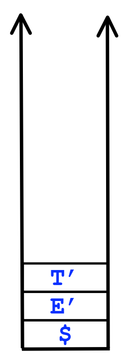
E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

Input:

(id*id)+id

Output:

E → T E'
T → F T'
F → (E)
E → T E'
T → F T'
F → id
T' → * F T'
F → id
T' → ε
E' → ε
T' → ε
E' → + T E'
T → F T'
F → id



(	id	*	id	)	+	id	$

Table [T', $] = T'→ε
Pop T'
Push <nothing>
Print T'→ε

	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)		

Example

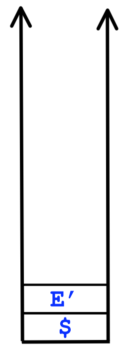
Input:

(id*id)+id

Output:

E → T E'
T → F T'
F → (E)
E → T E'
T → F T'
F → id
T' → * F T'
F → id
T' → ε
E' → ε
T' → ε
E' → + T E'
T → F T'
F → id
T' → ε

E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id



(	id	*	id	)	+	id	$

Table [T', $] = T'→ε
Pop T'
Push <nothing>
Print T'→ε

	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)		

Example

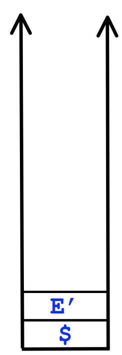
Input:

(id*id)+id

Output:

E → T E'
T → F T'
F → (E)
E → T E'
T → F T'
F → id
T' → * F T'
F → id
T' → ε
E' → ε
T' → ε
E' → + T E'
T → F T'
F → id
T' → ε

E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id



(	id	*	id	)	+	id	$

Table [E', $] = E'→ε
Pop E'
Push <nothing>
Print E'→ε

	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)		

Example

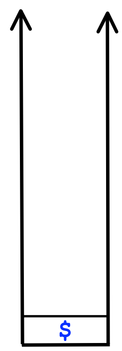
E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

Input:

(id*id)+id

Output:

E → T E'
T → F T'
F → (E)
E → T E'
T → F T'
F → id
T' → * F T'
F → id
T' → ε
E' → ε
T' → ε
E' → + T E'
T → F T'
F → id
T' → ε
E' → ε



(	id	*	id	)	+	id	$

Table [E', $] = E'→ε
Pop E'
Push <nothing>
Print E'→ε

	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)		

Example

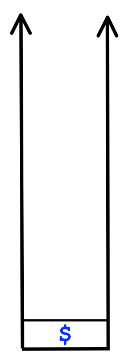
E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

Input:

(id*id)+id

Output:

E → T E'
T → F T'
F → (E)
E → T E'
T → F T'
F → id
T' → * F T'
F → id
T' → ε
E' → ε
T' → ε
E' → + T E'
T → F T'
F → id
T' → ε
E' → ε



(	id	*	id	)	+	id	$

Input symbol == $
Top of stack == $
Loop terminates with success

	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)		

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

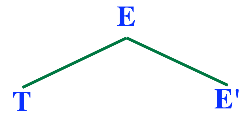
Reconstructing the Parse Tree

Input:

`(id*id)+id`

Output:

$E \rightarrow T\ E'$



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

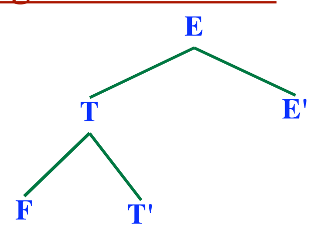
Reconstructing the Parse Tree

Input:

`(id*id)+id`

Output:

$E \rightarrow T\ E'$
$T \rightarrow F\ T'$



E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

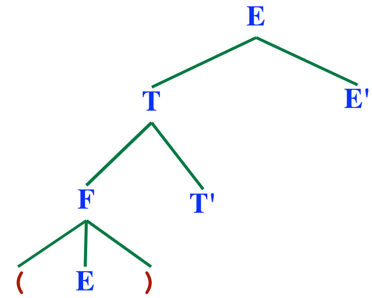
Reconstructing the Parse Tree

Input:

`(id*id)+id`

Output:

E → T E'
T → F T'
F → (E)



E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

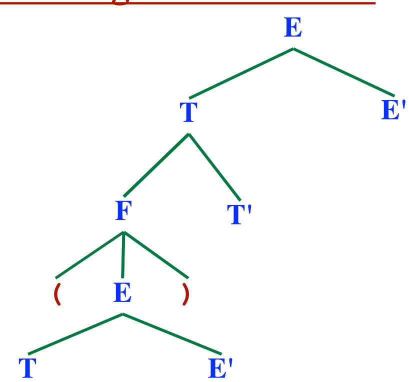
Reconstructing the Parse Tree

Input:

`(id*id)+id`

Output:

E → T E'
T → F T'
F → (E)
E → T E'



E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

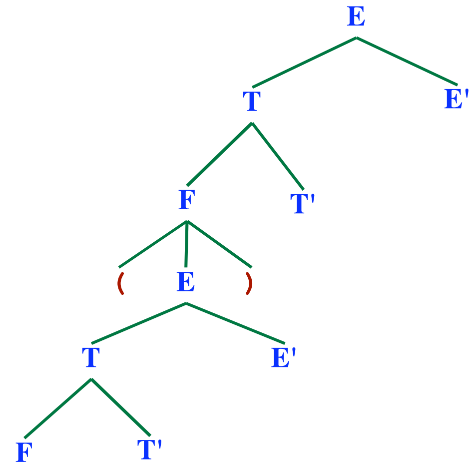
Reconstructing the Parse Tree

Input:

`(id*id)+id`

Output:

E → T E'
T → F T'
F → (E)
E → T E'
T → F T'



E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

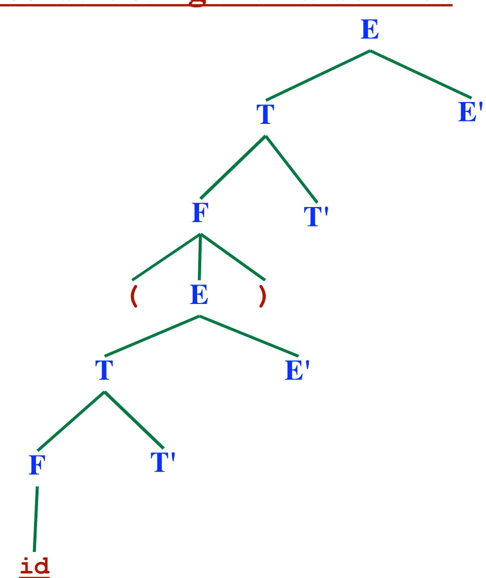
Reconstructing the Parse Tree

Input:

`(id*id)+id`

Output:

E → T E'
T → F T'
F → (E)
E → T E'
T → F T'
F → id



E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | <u>id</u>

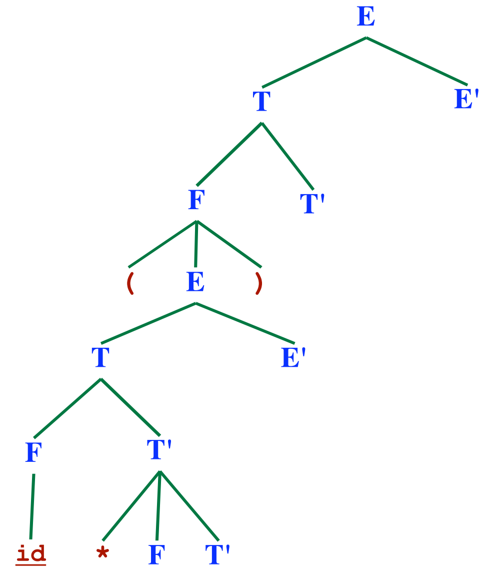
Reconstructing the Parse Tree

Input:

(id*id)+id

Output:

E → T E'
T → F T'
F → (E)
E → T E'
T → F T'
F → <u>id</u>
T' → * F T'



E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | <u>id</u>

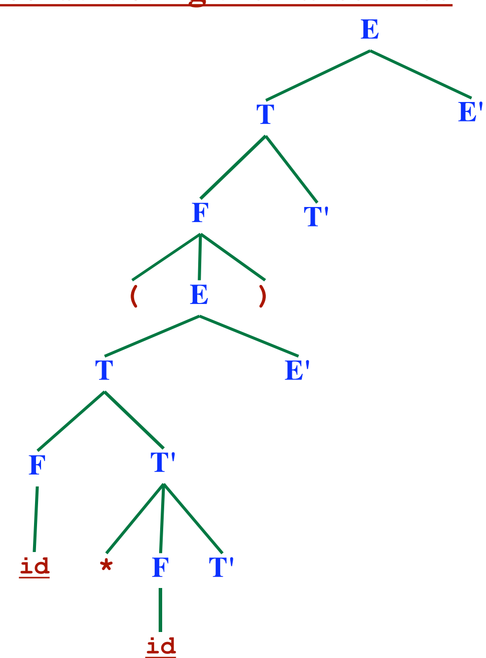
Reconstructing the Parse Tree

Input:

(id*id)+id

Output:

E → T E'
T → F T'
F → (E)
E → T E'
T → F T'
F → <u>id</u>
T' → * F T'
F → <u>id</u>



E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

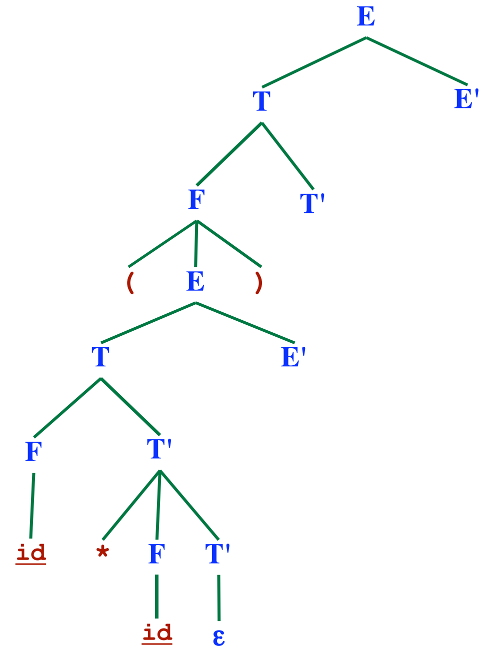
Reconstructing the Parse Tree

Input:

```
(id*id)+id
```

Output:

E → T E'
T → F T'
F → (E)
E → T E'
T → F T'
F → id
T' → * F T'
F → id
T' → ε



E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

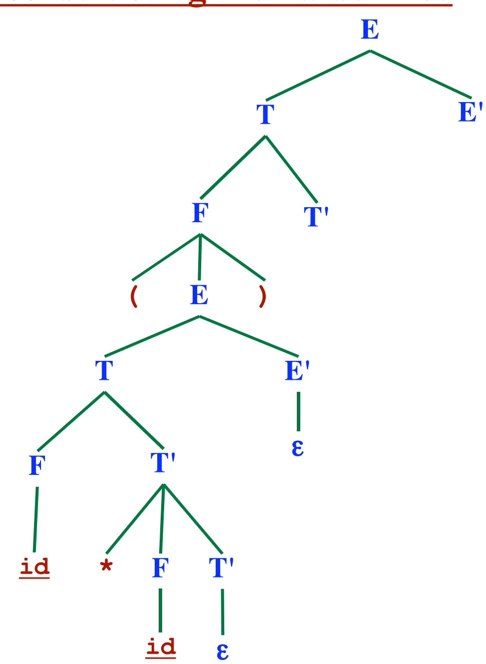
Reconstructing the Parse Tree

Input:

```
(id*id)+id
```

Output:

E → T E'
T → F T'
F → (E)
E → T E'
T → F T'
F → id
T' → * F T'
F → id
T' → ε
E' → ε



Reconstructing the Parse Tree

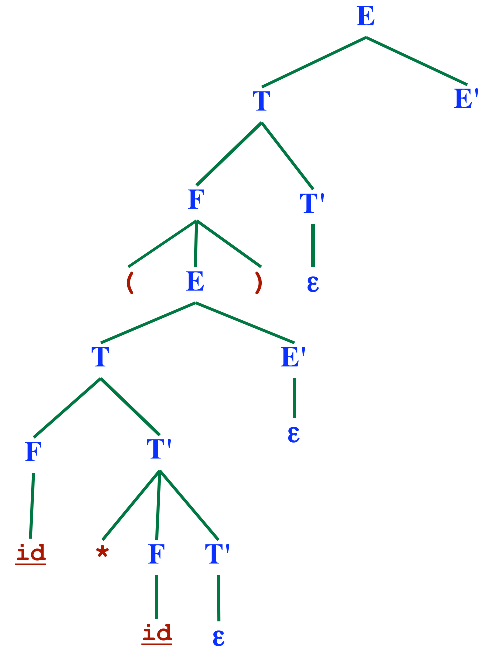
E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

Input:

(id*id)+id

Output:

E → T E'
T → F T'
F → (E)
E → T E'
T → F T'
F → id
T' → * F T'
F → id
T' → ε
E' → ε
T' → ε



Reconstructing the Parse Tree

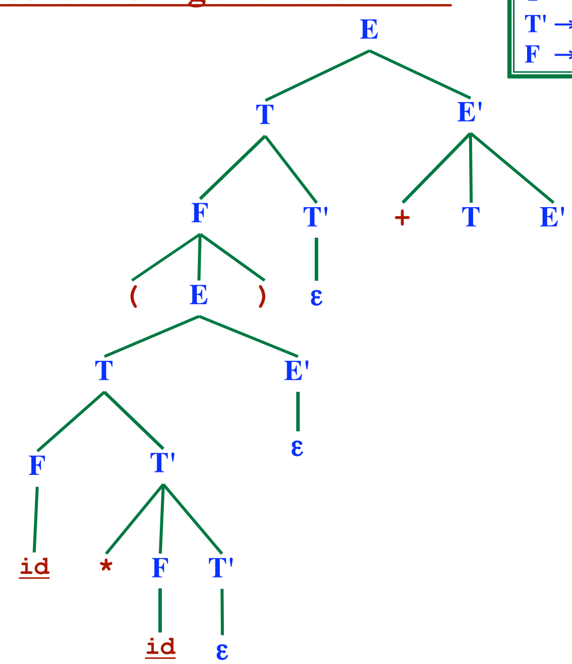
E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

Input:

(id*id)+id

Output:

E → T E'
T → F T'
F → (E)
E → T E'
T → F T'
F → id
T' → * F T'
F → id
T' → ε
E' → ε
T' → ε
E' → + T E'



E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

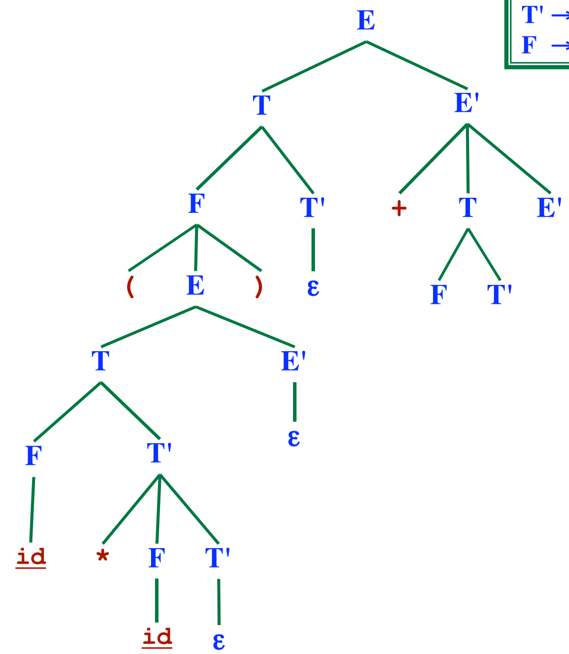
Reconstructing the Parse Tree

Input:

(id*id)+id

Output:

E → T E'
T → F T'
F → (E)
E → T E'
T → F T'
F → id
T' → * F T'
F → id
T' → ε
E' → ε
T' → ε
E' → + T E'
T → F T'



E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

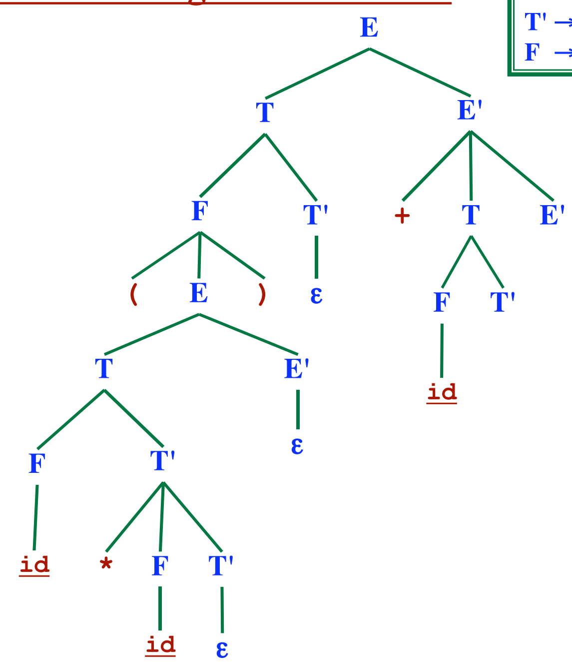
Reconstructing the Parse Tree

Input:

(id*id)+id

Output:

E → T E'
T → F T'
F → (E)
E → T E'
T → F T'
F → id
T' → * F T'
F → id
T' → ε
E' → ε
T' → ε
E' → + T E'
T → F T'
F → id



E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

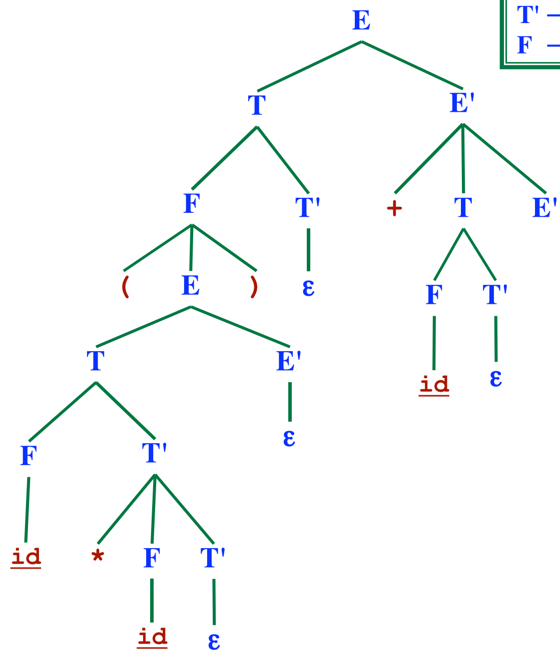
Reconstructing the Parse Tree

Input:

`(id*id)+id`

Output:

E → T E'
T → F T'
F → (E)
E → T E'
T → F T'
F → id
T' → * F T'
F → id
T' → ε
E' → ε
T' → ε
E' → + T E'
T → F T'
F → id
T' → ε



E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

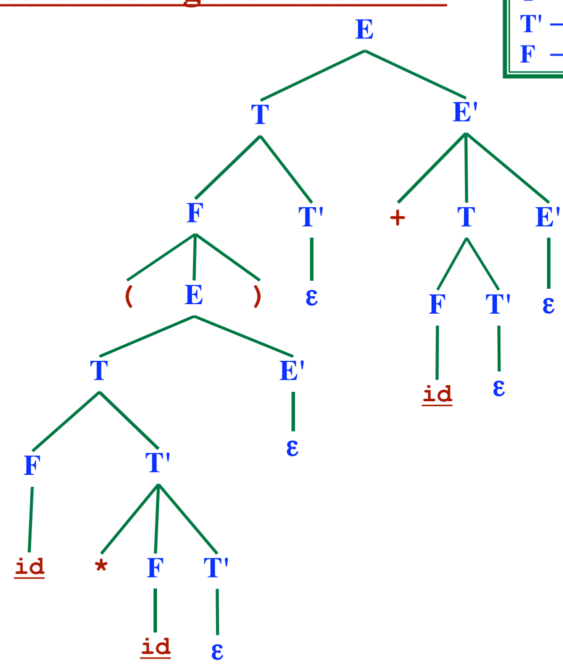
Reconstructing the Parse Tree

Input:

`(id*id)+id`

Output:

E → T E'
T → F T'
F → (E)
E → T E'
T → F T'
F → id
T' → * F T'
F → id
T' → ε
E' → ε
T' → ε
E' → + T E'
T → F T'
F → id
T' → ε
E' → ε



Reconstructing the Parse Tree

E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

Input:

`(id*id)+id`

Output:

E → T E'
T → F T'
F → (E)
E → T E'
T → F T'
F → id
T' → * F T'
F → id
T' → ε
E' → ε
T' → ε
E' → + T E'
T → F T'
F → id
T' → ε
E' → ε

Leftmost Derivation:

E
T E'
F T' E'
(E) T' E'
(T E') T' E'
(F T' E') T' E'
(id T' E') T' E'
(id * F T' E') T' E'
(id * id T' E') T' E'
(id * id E') T' E'
(id * id) T' E'
(id * id) E'
(id * id) + T E'
(id * id) + F T' E'
(id * id) + id T' E'
(id * id) + id E'
(id * id) + id

"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** | $\alpha \Rightarrow^* \mathbf{aw}$, plus ε if $\alpha \Rightarrow^* \varepsilon$ }

"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 | $\alpha \Rightarrow^*$ aw, plus ε if $\alpha \Rightarrow^* \varepsilon$ }

Example:

```
E  → T E'
E' → + T E' | ε
T  → F T'
T' → * F T' | ε
F  → ( E ) | id
```

FIRST (F) = ?

"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 | $\alpha \Rightarrow^*$ aw, plus ε if $\alpha \Rightarrow^* \varepsilon$ }

Example:

```
E  → T E'
E' → + T E' | ε
T  → F T'
T' → * F T' | ε
F  → ( E ) | id
```


FIRST (F) = { (, id }

FIRST (T') = ?

"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` | α ⇒* `aw`, plus ε if α ⇒* ε }

Example:

```
E  → T E'
E' → + T E' | ε
T  → F T'
T' → * F T' | ε
F  → ( E ) | id
```

FIRST (F) = { `(`, `id` }
FIRST (T') = { `*`, ε}
FIRST (T) = ?

"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` | α ⇒* `aw`, plus ε if α ⇒* ε }

Example:

```
E  → T E'
E' → + T E' | ε
T  → F T'
T' → * F T' | ε
F  → ( E ) | id
```

FIRST (F) = { `(`, `id` }
FIRST (T') = { `*`, ε}

FIRST (T) = { `(`, `id` }
FIRST (E') = ?

"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

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

Example:

E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

FIRST (F) = { (, id }
FIRST (T') = { *, ε}
FIRST (T) = { (, id }
FIRST (E') = { +, ε}
FIRST (E) = ?

"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

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

Example:

E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

FIRST (F) = { (, id }
FIRST (T') = { *, ε}
FIRST (T) = { (, id }
FIRST (E') = { +, ε}
FIRST (E) = { (, id }

To Compute the "FIRST" Function

For all symbols X in the grammar...

```
if X is a terminal then
  FIRST(X) = { X }

if X → ε is a rule then
  add ε to FIRST(X)

if X → Y1 Y2 Y3 ... YK is a rule then
  if a ∈ FIRST(Y1) then
    add a to FIRST(X)
  if ε ∈ FIRST(Y1) and a ∈ FIRST(Y2) then
    add a to FIRST(X)
  if ε ∈ FIRST(Y1) and ε ∈ FIRST(Y2) and a ∈ FIRST(Y3) then
    add a to FIRST(X)
  ...
  if ε ∈ FIRST(Yi) for all Yi then
    add ε to FIRST(X)
```

Repeat until nothing more can be added to any sets.

To Compute the FIRST($X_1X_2X_3...X_N$)

```
Result = {}
Add everything in FIRST(X1), except ε, to result
```

To Compute the FIRST($X_1X_2X_3...X_N$)

```
Result = {}
Add everything in FIRST(X1), except ε, to result
if ε ∈ FIRST(X1) then
  Add everything in FIRST(X2), except ε, to result
endIf
```


To Compute the FIRST($X_1X_2X_3...X_N$)

```
Result = {}
Add everything in FIRST(X1), except ε, to result
if ε ∈ FIRST(X1) then
  Add everything in FIRST(X2), except ε, to result
  if ε ∈ FIRST(X2) then
    Add everything in FIRST(X3), except ε, to result
  endIf
endIf
```

To Compute the FIRST($X_1X_2X_3...X_N$)

```
Result = {}
Add everything in FIRST(X1), except ε, to result
if ε ∈ FIRST(X1) then
  Add everything in FIRST(X2), except ε, to result
  if ε ∈ FIRST(X2) then
    Add everything in FIRST(X3), except ε, to result
    if ε ∈ FIRST(X3) then
      Add everything in FIRST(X4), except ε, to result









    endIf
  endIf
endIf
```


Syntax Analysis - Part 1

To Compute the FIRST($X_1X_2X_3...X_N$)

```
Result = {}
Add everything in FIRST(X1), except ε, to result
if ε ∈ FIRST(X1) then
  Add everything in FIRST(X2), except ε, to result
  if ε ∈ FIRST(X2) then
    Add everything in FIRST(X3), except ε, to result
    if ε ∈ FIRST(X3) then
      Add everything in FIRST(X4), except ε, to result
      ...
        if ε ∈ FIRST(XN-1) then
          Add everything in FIRST(XN), except ε, to result




        endIf
      ...
    endIf
  endIf
endIf
```

To Compute the FIRST($X_1X_2X_3...X_N$)

```
Result = {}
Add everything in FIRST(X1), except ε, to result
if ε ∈ FIRST(X1) then
  Add everything in FIRST(X2), except ε, to result
  if ε ∈ FIRST(X2) then
    Add everything in FIRST(X3), except ε, to result
    if ε ∈ FIRST(X3) then
      Add everything in FIRST(X4), except ε, to result
      ...
        if ε ∈ FIRST(XN-1) then
          Add everything in FIRST(XN), except ε, to result
          if ε ∈ FIRST(XN) then
            // Then X1 ⇒* ε, X2 ⇒* ε, X3 ⇒* ε, ... XN ⇒* ε
            Add ε to result
          endIf
        endIf
      ...
    endIf
  endIf
endIf
```


To Compute FOLLOW(A_i) for all Nonterminals in the Grammar

```
add $ to FOLLOW(S)
repeat
  if A→αBβ is a rule then
    add every terminal in FIRST(β) except ε to FOLLOW(B)
    if FIRST(β) contains ε then
      add everything in FOLLOW(A) to FOLLOW(B)
    endIf
  endIf
  if A→αB is a rule then
    add everything in FOLLOW(A) to FOLLOW(B)
  endIf
until We cannot add anything more
```

Example of FOLLOW Computation

Previously computed FIRST sets...

FIRST (F) = { (, id }
FIRST (T') = { *, ε}
FIRST (T) = { (, id }
FIRST (E') = { +, ε}
FIRST (E) = { (, id }

The FOLLOW sets...

FOLLOW (E) = { ?
FOLLOW (E') = { ?
FOLLOW (T) = { ?
FOLLOW (T') = { ?
FOLLOW (F) = { ?

E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

Example of FOLLOW Computation

Previously computed FIRST sets...

FIRST (F) = { (, id }
FIRST (T') = { *, ε}
FIRST (T) = { (, id }
FIRST (E') = { +, ε}
FIRST (E) = { (, id }

The FOLLOW sets...

FOLLOW (E) = {
FOLLOW (E') = {
FOLLOW (T) = {
FOLLOW (T') = {
FOLLOW (F) = {

E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

Add $ to FOLLOW(S)

Example of FOLLOW Computation

Previously computed FIRST sets...

FIRST (F) = { (, id }
FIRST (T') = { *, ε}
FIRST (T) = { (, id }
FIRST (E') = { +, ε}
FIRST (E) = { (, id }

The FOLLOW sets...

FOLLOW (E) = { $,
FOLLOW (E') = {
FOLLOW (T) = {
FOLLOW (T') = {
FOLLOW (F) = {

E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

Add $ to FOLLOW(S)

Syntax Analysis - Part 1

Example of FOLLOW Computation

Previously computed FIRST sets...

FIRST (F) = { (, id }
FIRST (T') = { *, ε}
FIRST (T) = { (, id }
FIRST (E') = { +, ε}
FIRST (E) = { (, id }

The FOLLOW sets...

FOLLOW (E) = { $,
FOLLOW (E') = {
FOLLOW (T) = {
FOLLOW (T') = {
FOLLOW (F) = {

E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

Look at rule
F → (E) | id
What can follow E?

Example of FOLLOW Computation

Previously computed FIRST sets...

FIRST (F) = { (, id }
FIRST (T') = { *, ε}
FIRST (T) = { (, id }
FIRST (E') = { +, ε}
FIRST (E) = { (, id }

The FOLLOW sets...

FOLLOW (E) = { $,)
FOLLOW (E') = {
FOLLOW (T) = {
FOLLOW (T') = {
FOLLOW (F) = {

E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

Look at rule
F → (E) | id
What can follow E?

Example of FOLLOW Computation

Previously computed FIRST sets...

FIRST (F) = { (, id }
FIRST (T') = { *, ε}
FIRST (T) = { (, id }
FIRST (E') = { +, ε}
FIRST (E) = { (, id }

The FOLLOW sets...

FOLLOW (E) = { $,)
FOLLOW (E') = {
FOLLOW (T) = {
FOLLOW (T') = {
FOLLOW (F) = {

E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

Look at rule
E → T E'
Whatever can follow E can also follow E'

Example of FOLLOW Computation

E	→	T E'
E'	→	+ T E' | ε
T	→	F T'
T'	→	* F T' | ε
F	→	(E) | id

Previously computed FIRST sets...

FIRST (F) = { (, id }
FIRST (T') = { *, ε}
FIRST (T) = { (, id }
FIRST (E') = { +, ε}
FIRST (E) = { (, id }

The FOLLOW sets...

FOLLOW (E) = { $,)
FOLLOW (E') = { $,)
FOLLOW (T) = {
FOLLOW (T') = {
FOLLOW (F) = {

Look at rule
E → T E'
Whatever can follow E
can also follow E'

Syntax Analysis - Part 1

Example of FOLLOW Computation

E	→	T E'
E'	→	+ T E' | ε
T	→	F T'
T'	→	* F T' | ε
F	→	(E) | id

Previously computed FIRST sets...

FIRST (F) = { (, id }
FIRST (T') = { *, ε}
FIRST (T) = { (, id }
FIRST (E') = { +, ε}
FIRST (E) = { (, id }

The FOLLOW sets...

FOLLOW (E) = { $,)
FOLLOW (E') = { $,)
FOLLOW (T) = {
FOLLOW (T') = {
FOLLOW (F) = {

Look at rule
$E'_0 \rightarrow + T E'_1$
Whatever is in FIRST(E'_1)
can follow T

Example of FOLLOW Computation

E	→	T E'
E'	→	+ T E' | ε
T	→	F T'
T'	→	* F T' | ε
F	→	(E) | id

Previously computed FIRST sets...

FIRST (F) = { (, id }
FIRST (T') = { *, ε}
FIRST (T) = { (, id }
FIRST (E') = { +, ε}
FIRST (E) = { (, id }

The FOLLOW sets...

FOLLOW (E) = { $,)
FOLLOW (E') = { $,)
FOLLOW (T) = { +,
FOLLOW (T') = {
FOLLOW (F) = {

Look at rule

$E'_0 \rightarrow + T E'_1$

Whatever is in FIRST(E'_1) can follow T

Syntax Analysis - Part 1

Example of FOLLOW Computation

E	→	T E'
E'	→	+ T E' | ε
T	→	F T'
T'	→	* F T' | ε
F	→	(E) | id

Previously computed FIRST sets...

FIRST (F) = { (, id }
FIRST (T') = { *, ε}
FIRST (T) = { (, id }
FIRST (E') = { +, ε}
FIRST (E) = { (, id }

The FOLLOW sets...

FOLLOW (E) = { $,)
FOLLOW (E') = { $,)
FOLLOW (T) = { +,
FOLLOW (T') = {
FOLLOW (F) = {

Look at rule

$T'_0 \rightarrow * F T'_1$

Whatever is in FIRST(T'_1) can follow F

Example of FOLLOW Computation

Previously computed FIRST sets...

FIRST (F) = { `(`, `id` }
FIRST (T') = { `*`, ε}
FIRST (T) = { `(`, `id` }
FIRST (E') = { `+`, ε}
FIRST (E) = { `(`, `id` }

The FOLLOW sets...

FOLLOW (E) = { `$`, `)`
FOLLOW (E') = { `$`, `)`
FOLLOW (T) = { `+`,
FOLLOW (T') = {
FOLLOW (F) = { `*`,

E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

Look at rule

$T'_0 \rightarrow$ * $F\ T'_1$

Whatever is in FIRST(T'_1) can follow F

Syntax Analysis - Part 1

Example of FOLLOW Computation

Previously computed FIRST sets...

FIRST (F) = { `(`, `id` }
FIRST (T') = { `*`, ε}
FIRST (T) = { `(`, `id` }
FIRST (E') = { `+`, ε}
FIRST (E) = { `(`, `id` }

The FOLLOW sets...

FOLLOW (E) = { `$`, `)`
FOLLOW (E') = { `$`, `)`
FOLLOW (T) = { `+`,
FOLLOW (T') = {
FOLLOW (F) = { `*`,

E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

Look at rule

$E'_0 \rightarrow$ + $T\ E'_1$

Since E'_1 can go to ε

i.e., ε ∈ FIRST(E')

Everything in FOLLOW(E'_0) can follow T

Example of FOLLOW Computation

E	→	T E'
E'	→	+ T E' | ε
T	→	F T'
T'	→	* F T' | ε
F	→	(E) | id

Previously computed FIRST sets...

FIRST (F) = { (, id }
FIRST (T') = { *, ε}
FIRST (T) = { (, id }
FIRST (E') = { +, ε}
FIRST (E) = { (, id }

The FOLLOW sets...

FOLLOW (E) = { $,)
FOLLOW (E') = { $,)
FOLLOW (T) = { +, $,)
FOLLOW (T') = {
FOLLOW (F) = { *,

Look at rule

$E'_0 \rightarrow + T E'_1$

Since E'_1 **can go to ε**

i.e., ε ∈ FIRST(E')

Everything in FOLLOW(E'_0**) can follow T**

Example of FOLLOW Computation

E	→	T E'
E'	→	+ T E' | ε
T	→	F T'
T'	→	* F T' | ε
F	→	(E) | id

Previously computed FIRST sets...

FIRST (F) = { (, id }
FIRST (T') = { *, ε}
FIRST (T) = { (, id }
FIRST (E') = { +, ε}
FIRST (E) = { (, id }

The FOLLOW sets...

FOLLOW (E) = { $,)
FOLLOW (E') = { $,)
FOLLOW (T) = { +, $,)
FOLLOW (T') = {
FOLLOW (F) = { *,

Look at rule

T → F T'

Whatever can follow T can also follow T'

Example of FOLLOW Computation

Previously computed FIRST sets...

FIRST (F) = { (, id }
FIRST (T') = { *, ε}
FIRST (T) = { (, id }
FIRST (E') = { +, ε}
FIRST (E) = { (, id }

The FOLLOW sets...

FOLLOW (E) = { $,)
FOLLOW (E') = { $,)
FOLLOW (T) = { +, $,)
FOLLOW (T') = { +, $,)
FOLLOW (F) = { *,

E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

Look at rule
T → F T'
Whatever can follow T
can also follow T'

Example of FOLLOW Computation

Previously computed FIRST sets...

FIRST (F) = { (, id }
FIRST (T') = { *, ε}
FIRST (T) = { (, id }
FIRST (E') = { +, ε}
FIRST (E) = { (, id }

The FOLLOW sets...

FOLLOW (E) = { $,)
FOLLOW (E') = { $,)
FOLLOW (T) = { +, $,)
FOLLOW (T') = { +, $,)
FOLLOW (F) = { *,

E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

Look at rule
$T'_0 \rightarrow * F T'_1$
Since T'_1 can go to ε
i.e., ε ∈ FIRST(T')
Everything in FOLLOW(T'_0)
can follow F

Example of FOLLOW Computation

Previously computed FIRST sets...

FIRST (F) = { (, id }
FIRST (T') = { *, ε}
FIRST (T) = { (, id }
FIRST (E') = { +, ε}
FIRST (E) = { (, id }

The FOLLOW sets...

FOLLOW (E) = { $,)
FOLLOW (E') = { $,)
FOLLOW (T) = { +, $,)
FOLLOW (T') = { +, $,)
FOLLOW (F) = { *, +, $,)

E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

Look at rule

$T'_0 \rightarrow$ * F T'_1

Since T'_1 can go to ε

i.e., ε ∈ FIRST(T')

Everything in FOLLOW(T'_0) can follow F

Example of FOLLOW Computation

Previously computed FIRST sets...

FIRST (F) = { (, id }
FIRST (T') = { *, ε}
FIRST (T) = { (, id }
FIRST (E') = { +, ε}
FIRST (E) = { (, id }

The FOLLOW sets...

FOLLOW (E) = { $,) }
FOLLOW (E') = { $,) }
FOLLOW (T) = { +, $,) }
FOLLOW (T') = { +, $,) }
FOLLOW (F) = { *, +, $,) }

E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

Nothing more can be added.

Building the Predictive Parsing Table

The Main Idea:

Assume we're looking for an A
i.e., A is on the stack top.
Assume b is the current input symbol.

Building the Predictive Parsing Table

The Main Idea:

Assume we're looking for an A
i.e., A is on the stack top.
Assume b is the current input symbol.

If $A \to \alpha$ is a rule and b is in FIRST(α)
then expand A using the $A \to \alpha$ rule!

Building the Predictive Parsing Table

The Main Idea:

Assume we're looking for an A
i.e., A is on the stack top.
Assume b is the current input symbol.

If $A \rightarrow \alpha$ is a rule and b is in FIRST(α)
then expand A using the $A \rightarrow \alpha$ rule!

What if ε is in FIRST(α)? [i.e., $\alpha \Rightarrow^* \varepsilon$]
If b is in FOLLOW(A)
then expand A using the $A \rightarrow \alpha$ rule!

Building the Predictive Parsing Table

The Main Idea:

Assume we're looking for an A
i.e., A is on the stack top.
Assume b is the current input symbol.

If $A \rightarrow \alpha$ is a rule and b is in FIRST(α)
then expand A using the $A \rightarrow \alpha$ rule!

What if ε is in FIRST(α)? [i.e., $\alpha \Rightarrow^* \varepsilon$]
If b is in FOLLOW(A)
then expand A using the $A \rightarrow \alpha$ rule!

If ε is in FIRST(α) and \$ is the current input symbol
then if \$ is in FOLLOW(A)
then expand A using the $A \rightarrow \alpha$ rule!

Example: The "Dangling Else" Grammar

1.	S	→	**if** E **then** S S'
2.	S	→	**otherStmt**
3.	S'	→	**else** S
4.	S'	→	ε
5.	E	→	**boolExpr**

"if b then if b then otherStmt else otherStmt"

Example: The "Dangling Else" Grammar

1.	S	→	**i** E **t** S S'
2.	S	→	**o**
3.	S'	→	**e** S
4.	S'	→	ε
5.	E	→	**b**

←

1.	S	→	**if** E **then** S S'
2.	S	→	**otherStmt**
3.	S'	→	**else** S
4.	S'	→	ε
5.	E	→	**boolExpr**

i b t i b t o e o ⇐ "if b then if b then otherStmt else otherStmt"

Example: The "Dangling Else" Grammar

1. S → **i** E **t** S S'
2. S → **o**
3. S' → **e** S
4. S' → ε
5. E → **b**

i b t i b t o e o

Example: The "Dangling Else" Grammar

1. S → **i** E **t** S S'
2. S → **o**
3. S' → **e** S
4. S' → ε
5. E → **b**

i b t i b t o e o

FIRST(S) = { **i**, **o** }
FIRST(S') = { **e**, ε }
FIRST(E) = { **b** }

FOLLOW(S) = { **e**, **$** }
FOLLOW(S') = { **e**, **$** }
FOLLOW(E) = { **t** }

Example: The "Dangling Else" Grammar

1. S → i E t S S'
2. S → o
3. S' → e S
4. S' → ε
5. E → b

Look at Rule 1: S → i E t S S'
If we are looking for an S
and the next symbol is in FIRST(i E t S S')...
Add that rule to the table

i b t i b t o e o

FIRST(S) = { i, o }   FOLLOW(S) = { e, $ }
FIRST(S') = { e, ε }   FOLLOW(S') = { e, $ }
FIRST(E) = { b }   FOLLOW(E) = { t }

	o	b	e	i	t	$
S						
S'						
E						

Example: The "Dangling Else" Grammar

1. S → i E t S S'
2. S → o
3. S' → e S
4. S' → ε
5. E → b

Look at Rule 1: S → i E t S S'
If we are looking for an S
and the next symbol is in FIRST(i E t S S')...
Add that rule to the table

i b t i b t o e o

FIRST(S) = { i, o }   FOLLOW(S) = { e, $ }
FIRST(S') = { e, ε }   FOLLOW(S') = { e, $ }
FIRST(E) = { b }   FOLLOW(E) = { t }

	o	b	e	i	t	$
S				S → iEtSS'		
S'						
E						

Example: The "Dangling Else" Grammar

1. S → i E t S S'
2. S → o
3. S' → e S
4. S' → ε
5. E → b

Look at Rule 2: S → o
If we are looking for an S
and the next symbol is in FIRST(o)...
Add that rule to the table

i b t i b t o e o

FIRST(S) = { i, o }   FOLLOW(S) = { e, $ }
FIRST(S') = { e, ε }   FOLLOW(S') = { e, $ }
FIRST(E) = { b }   FOLLOW(E) = { t }

	o	b	e	i	t	$
S				S → iEtSS'		
S'						
E						

Syntax Analysis - Part 1

Example: The "Dangling Else" Grammar

1. S → i E t S S'
2. S → o
3. S' → e S
4. S' → ε
5. E → b

Look at Rule 2: S → o
If we are looking for an S
and the next symbol is in FIRST(o)...
Add that rule to the table

i b t i b t o e o

FIRST(S) = { i, o }   FOLLOW(S) = { e, $ }
FIRST(S') = { e, ε }   FOLLOW(S') = { e, $ }
FIRST(E) = { b }   FOLLOW(E) = { t }

	o	b	e	i	t	$
S	S → o			S → iEtSS'		
S'						
E						

Example: The "Dangling Else" Grammar

1. S → i E t S S'
2. S → o
3. S' → e S
4. S' → ε
5. E → b

Look at Rule 5: E → b
If we are looking for an E
and the next symbol is in FIRST(b)...
Add that rule to the table

i b t i b t o e o

FIRST(S) = { i, o }   FOLLOW(S) = { e, $ }
FIRST(S') = { e, ε }   FOLLOW(S') = { e, $ }
FIRST(E) = { b }   FOLLOW(E) = { t }

	o	b	e	i	t	$
S	S → o			S → iEtSS'		
S'						
E						

Example: The "Dangling Else" Grammar

1. S → i E t S S'
2. S → o
3. S' → e S
4. S' → ε
5. E → b

Look at Rule 5: E → b
If we are looking for an E
and the next symbol is in FIRST(b)...
Add that rule to the table

i b t i b t o e o

FIRST(S) = { i, o }   FOLLOW(S) = { e, $ }
FIRST(S') = { e, ε }   FOLLOW(S') = { e, $ }
FIRST(E) = { b }   FOLLOW(E) = { t }

	o	b	e	i	t	$
S	S → o			S → iEtSS'		
S'						
E		E → b				

Example: The "Dangling Else" Grammar

1. S → i E t S S'
2. S → o
3. S' → e S
4. S' → ε
5. E → b

i b t i b t o e o

Look at Rule 3: S' → e S
If we are looking for an S'
and the next symbol is in FIRST(e S)...
Add that rule to the table

FIRST(S) = { i, o } FOLLOW(S) = { e, $ }
FIRST(S') = { e, ε } FOLLOW(S') = { e, $ }
FIRST(E) = { b } FOLLOW(E) = { t }

	o	b	e	i	t	$
S	S → o			S → iEtSS'		
S'						
E		E → b				

Example: The "Dangling Else" Grammar

1. S → i E t S S'
2. S → o
3. S' → e S
4. S' → ε
5. E → b

i b t i b t o e o

Look at Rule 3: S' → e S
If we are looking for an S'
and the next symbol is in FIRST(e S)...
Add that rule to the table

FIRST(S) = { i, o } FOLLOW(S) = { e, $ }
FIRST(S') = { e, ε } FOLLOW(S') = { e, $ }
FIRST(E) = { b } FOLLOW(E) = { t }

	o	b	e	i	t	$
S	S → o			S → iEtSS'		
S'			S' → e S			
E		E → b				

Example: The "Dangling Else" Grammar

1. S → i E t S S'
2. S → o
3. S' → e S
4. S' → ε
5. E → b

Look at Rule 4: S' → ε
If we are looking for an S'
and ε ∈ FIRST(rhs)...
Then if $ ∈ FOLLOW(S')...
Add that rule under $

i b t i b t o e o

FIRST(S) = { i, o }
FIRST(S') = { e, ε }
FIRST(E) = { b }

FOLLOW(S) = { e, $ }
FOLLOW(S') = { e, $ }
FOLLOW(E) = { t }

	o	b	e	i	t	$
S	S → o			S → iEtSS'		
S'			S' → e S			
E		E → b				

Example: The "Dangling Else" Grammar

1. S → i E t S S'
2. S → o
3. S' → e S
4. S' → ε
5. E → b

Look at Rule 4: S' → ε
If we are looking for an S'
and ε ∈ FIRST(rhs)...
Then if $ ∈ FOLLOW(S')...
Add that rule under $

i b t i b t o e o

FIRST(S) = { i, o }
FIRST(S') = { e, ε }
FIRST(E) = { b }

FOLLOW(S) = { e, $ }
FOLLOW(S') = { e, $ }
FOLLOW(E) = { t }

	o	b	e	i	t	$
S	S → o			S → iEtSS'		
S'			S' → e S			S' → ε
E		E → b				

Example: The "Dangling Else" Grammar

1. S → i E t S S'
2. S → o
3. S' → e S
4. S' → ε
5. E → b

Look at Rule 4: S' → ε
If we are looking for an S'
and ε ∈ FIRST(rhs)...
Then if e ∈ FOLLOW(S')...
Add that rule under e

i b t i b t o e o

FIRST(S) = { i, o }   FOLLOW(S) = { e, $ }
FIRST(S') = { e, ε }   FOLLOW(S') = { e, $ }
FIRST(E) = { b }   FOLLOW(E) = { t }

	o	b	e	i	t	$
S	S → o			S → iEtSS'		
S'			S' → e S			S' → ε
E		E → b				

Example: The "Dangling Else" Grammar

1. S → i E t S S'
2. S → o
3. S' → e S
4. S' → ε
5. E → b

Look at Rule 4: S' → ε
If we are looking for an S'
and ε ∈ FIRST(rhs)...
Then if e ∈ FOLLOW(S')...
Add that rule under e

i b t i b t o e o

FIRST(S) = { i, o }   FOLLOW(S) = { e, $ }
FIRST(S') = { e, ε }   FOLLOW(S') = { e, $ }
FIRST(E) = { b }   FOLLOW(E) = { t }

	o	b	e	i	t	$
S	S → o			S → iEtSS'		
S'			S' → e S S' → ε			S' → ε
E		E → b				

Example: The "Dangling Else" Grammar

1. S → **i** E **t** S S'
2. S → **o**
3. S' → **e** S
4. S' → ε
5. E → **b**

CONFLICT!
Two rules in one table entry.

i b t i b t o e o

FIRST(S) = { **i**, **o** } FOLLOW(S) = { **e**, **$** }
FIRST(S') = { **e**, ε } FOLLOW(S') = { **e**, **$** }
FIRST(E) = { **b** } FOLLOW(E) = { **t** }

	o	**b**	**e**	**i**	**t**	**$**
S	S → **o**			S → **i**E**t**SS'		
S'			S' → **e** S S' → ε			S' → ε
E		E → **b**				

Syntax Analysis - Part 1

Example: The "Dangling Else" Grammar

1. S → **i** E **t** S S'
2. S → **o**
3. S' → **e** S
4. S' → ε
5. E → **b**

CONFLICT!
Two rules in one table entry.
The grammar is not LL(1)!

i b t i b t o e o

FIRST(S) = { **i**, **o** } FOLLOW(S) = { **e**, **$** }
FIRST(S') = { **e**, ε } FOLLOW(S') = { **e**, **$** }
FIRST(E) = { **b** } FOLLOW(E) = { **t** }

	o	**b**	**e**	**i**	**t**	**$**
S	S → **o**			S → **i**E**t**SS'		
S'			S' → **e** S S' → ε			S' → ε
E		E → **b**				

Algorithm to Build the Table

Input: Grammar G
Output: Parsing Table, such that **TABLE**[A,b] = Rule to use or "ERROR/Blank"

Syntax Analysis - Part 1

Algorithm to Build the Table

Input: Grammar G
Output: Parsing Table, such that **TABLE**[A,b] = Rule to use or "ERROR/Blank"

```
Compute FIRST and FOLLOW sets
```

Algorithm to Build the Table

Input: Grammar G
Output: Parsing Table, such that **TABLE[A,b]** = Rule to use or "ERROR/Blank"

```
Compute FIRST and FOLLOW sets
for each rule A→α do
  for each terminal b in FIRST(α) do
    add A→α to TABLE[A,b]
  endFor
endFor
```


Algorithm to Build the Table

Input: Grammar G
Output: Parsing Table, such that **TABLE[A,b]** = Rule to use or "ERROR/Blank"

```
Compute FIRST and FOLLOW sets
for each rule A→α do
  for each terminal b in FIRST(α) do
    add A→α to TABLE[A,b]
  endFor
  if ε is in FIRST(α) then
    for each terminal b in FOLLOW(A) do
      add A→α to TABLE[A,b]
    endFor



  endIf
endFor
```

Algorithm to Build the Table

Input: Grammar G
Output: Parsing Table, such that **TABLE[A,b]** = Rule to use or "ERROR/Blank"

```
Compute FIRST and FOLLOW sets
for each rule A→α do
  for each terminal b in FIRST(α) do
    add A→α to TABLE[A,b]
  endFor
  if ε is in FIRST(α) then
    for each terminal b in FOLLOW(A) do
      add A→α to TABLE[A,b]
    endFor
    if $ is in FOLLOW(A) then
      add A→α to TABLE[A,$]
    endIf
  endIf
endFor
```


Algorithm to Build the Table

Input: Grammar G
Output: Parsing Table, such that **TABLE[A,b]** = Rule to use or "ERROR/Blank"

```
Compute FIRST and FOLLOW sets
for each rule A→α do
  for each terminal b in FIRST(α) do
    add A→α to TABLE[A,b]
  endFor
  if ε is in FIRST(α) then
    for each terminal b in FOLLOW(A) do
      add A→α to TABLE[A,b]
    endFor
    if $ is in FOLLOW(A) then
      add A→α to TABLE[A,$]
    endIf
  endIf
endFor
TABLE[A,b] is undefined?  Then set TABLE[A,b] to "error"
```

Algorithm to Build the Table

Input: Grammar G
Output: Parsing Table, such that TABLE[A,b] = Rule to use or "ERROR/Blank"

```
Compute FIRST and FOLLOW sets
for each rule A→α do
  for each terminal b in FIRST(α) do
    add A→α to TABLE[A,b]
  endFor
  if ε is in FIRST(α) then
    for each terminal b in FOLLOW(A) do
      add A→α to TABLE[A,b]
    endFor
    if $ is in FOLLOW(A) then
      add A→α to TABLE[A,$]
    endIf
  endIf
endFor
TABLE[A,b] is undefined?  Then set TABLE[A,b] to "error"
TABLE[A,b] is multiply defined?
```

The algorithm fails!!! Grammar G is not LL(1)!!!

LL(1) Grammars

Using only one symbol of look-ahead
Find Leftmost derivation
Scanning input left-to-right

LL(1) grammars
- Are never ambiguous.
- Will never have left recursion.

Furthermore...

If we are looking for an "A" and the next symbol is "b",
Then only one production must be possible.

More Precisely...

If $A \to \alpha$ and $A \to \beta$ are two rules
If $\alpha \Rightarrow^* \underline{a}...$ and $\beta \Rightarrow^* \underline{b}...$
then we require $\underline{a} \neq \underline{b}$
(i.e., FIRST(α) and FIRST(β) must not intersect)

If $\alpha \Rightarrow^* \varepsilon$
then $\beta \Rightarrow^* \varepsilon$ must not be possible.
(i.e., only one alternative can derive ε.)
If $\alpha \Rightarrow^* \varepsilon$ and $\beta \Rightarrow^* \underline{b}...$
then $\underline{b}$ must not be in FOLLOW(A)

Error Recovery

We have an error whenever...

- Stacktop is a terminal, but stacktop ≠ input symbol
- Stacktop is a nonterminal but TABLE[A,b] is empty

Options

1. Skip over input symbols, until we can resume parsing
   Corresponds to ignoring tokens
2. Pop stack, until we can resume parsing
   Corresponds to inserting missing material
3. Some combination of 1 and 2
4. "Panic Mode" - Use Synchronizing tokens
   - Identify a set of synchronizing tokens.
   - Skip over tokens until we are positioned on a synchronizing token.
   - Pop stack until we can resume parsing.

Option 1: Skip Input Symbols

Example:

Decided to use rule

$S \rightarrow$ **IF** E **THEN** S **ELSE** S **END**

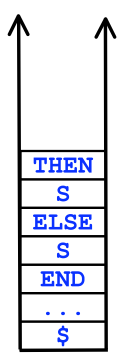
Stack tells us what we are expecting next in the input.

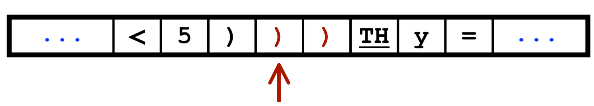
We've already gotten **IF** and E

Assume there are extra tokens in the input.

`if (x<5))) then y = 7; ...`

A syntax error occurs here.





We want to skip tokens until we can resume parsing.

Option 2: Pop The Stack

Example:

Decided to use rules

S → **IF** E **THEN** S **ELSE** S **END**

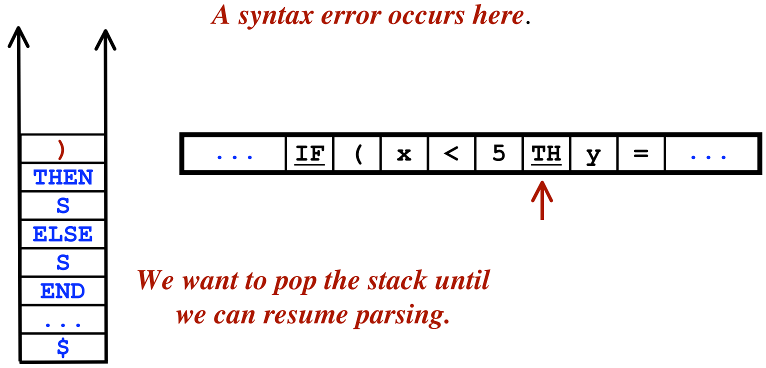
E → **(** E **)**

We've already gotten `if (` E

Assume there are missing tokens.

`if (x < 5 then y = 7;...`

A syntax error occurs here.



We want to pop the stack until we can resume parsing.

Panic Mode Recovery

The "***Synchronizing Set***" of tokens

... is determined by the compiler writer beforehand

Example: { SEMI-COLON, RIGHT-BRACE }

Skip input symbols until we find something in the synchronizing set.

Idea:

Look at the non-terminal on the stack top.
Choose the synchronizing set based on this non-terminal.
Assume A is on the stack top
Let SynchSet = FOLLOW(A)
Skip tokens until we see something in FOLLOW(A)
Pop A from the stack.
Should be able to keep going.

Idea:

Look at the non-terminals in the stack (e.g., A, B, C, ...)
Include FIRST(A), FIRST(B), FIRST(C), ... in the SynchSet.
Skip tokens until we see something in FIRST(A), FIRST(B), FIRST(C), ...
Pop stack until **NextToken** ∈ FIRST(**NonTerminalOnStackTop**)

Error Recovery - Table Entries

Each blank entry in the table indicates an error.
Tailor the error recovery for each possible error.
Fill the blank entry with an error routine.
The error routine will tell what to do.

Error Recovery - Table Entries

Each blank entry in the table indicates an error.
Tailor the error recovery for each possible error.
Fill the blank entry with an error routine.
The error routine will tell what to do.

Example:

	id	SEMI	RPAREN	LPAREN	...	$
E			E4			
E'			E5			
...						

Error Recovery - Table Entries

Each blank entry in the table indicates an error.
Tailor the error recovery for each possible error.
Fill the blank entry with an error routine.
The error routine will tell what to do.

Example:

	id	SEMI	RPAREN	LPAREN	...	$
E			E4			
E'			E5			
...						

Error-Handling Code

```
...
E4:
        SynchSet = { SEMI, IF, THEN }
        SkipTokensTo (SynchSet)
        Print ("Unexpected right paren")
        Pop stack
        break
E5:     ...
...
```

Choose the SynchSet based on the particular error