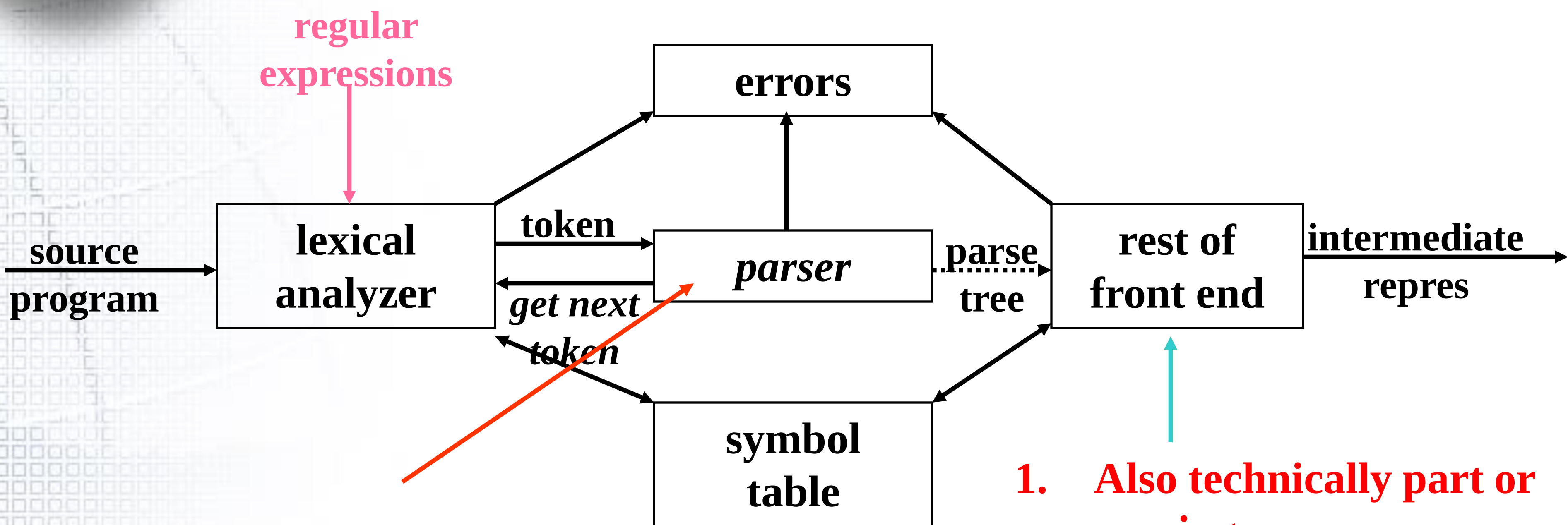
Module 3
Syntax Analysis

Syntax Analysis - Parsing

- An overview of parsing :
  - Functions & Responsibilities
- Context Free Grammars
  - Concepts & Terminology
- Writing and Designing Grammars
- Resolving Grammar Problems / Difficulties
- Top-Down Parsing
  - Recursive Descent & Predictive LL
- Bottom-Up Parsing
  - LR & LALR
- Concluding Remarks/Looking Ahead

Parsing During Compilation



1. Also technically part or parsing
2. Includes augmenting info on tokens in source, type checking, semantic analysis

- produces a parse tree
- syntactic errors and recovery
- recognize correct syntax
- report errors

Parsing Responsibilities

Syntax Error Identification / Handling

Recall typical error types:

- Lexical : Misspellings
- Syntactic : Omission, wrong order of tokens
- Semantic : Incompatible types
- Logical : Infinite loop / recursive call

Majority of error processing occurs during syntax analysis

NOTE: Not all errors are identifiable !!

Key Issues – Error Processing

1. **Detecting errors**
2. **Finding position at which they occur**
3. **Clear / accurate presentation**
4. **Recover (pass over) to continue and find later errors**
5. **Don’t impact compilation of “correct” programs**

What are some Typical Errors ?

```
#include<stdio.h>
int f1(int v)
{     int i,j=0;
      for (i=1;i<5;i++)
      {   j=v+f2(i) }
      return j; }
int f2(int u)
{     int j;
      j=u+f1(u*u);
      return j; }
int main()
{     int i,j=0;
            for (i=1;i<10;i++)
            {     j=j+i*i  printf(“%d\n”,i);        }
      printf("%d\n",f1(j));
      return 0;
}
```

As reported by MS VC++

'f2' undefined;
syntax error : missing ';' before '}‘
syntax error : missing ';' before identifier 'printf'

Which are “easy” to recover from? Which are “hard” ?

Error Recovery Strategies

Panic Mode – Discard tokens until a "synchronous" token is found (end, ";", "}", etc.)

-- Decision of designer

-- Problems:

skip input $\Rightarrow$ miss declaration – causing more errors

$\Rightarrow$ miss errors in skipped material

-- Advantages:

simple $\Rightarrow$ suited to 1 error per statement

Phrase Level – Local correction on input

-- "," $\Rightarrow$ ";" – Delete "," – insert ";"

-- Also decision of designer

-- Not suited to all situations

-- Used in conjunction with panic mode to allow less input to be skipped

Error Recovery Strategies – (2)

Error Productions:

--Augment grammar with rules

-- Augment grammar used for parser construction / generation

-- example: add a rule for := in C assignment statements

Report error but continue compile

-- Self correction + diagnostic messages

Global Correction:

-- Adding / deleting / replacing symbols may do many changes !

-- Algorithms available to minimize changes costly - key issues

Motivating Grammars

- Regular Expressions
  - → Basis of lexical analysis
  - → Represent regular languages
- Context Free Grammars
  - → Basis of parsing
  - → Represent language constructs

Context Free Grammars : Concepts & Terminology

Definition: A Context Free Grammar, CFG, is described by T, NT, S, PR, where:

T: **Terminals / tokens of the language**

NT: **Non-terminals to denote sets of strings generated by the grammar & in the language**

S: **Start symbol, S$\in$NT, which defines all strings of the language**

PR: **Production rules to indicate how T and NT are combined to generate valid strings of the language.**

PR: NT $\rightarrow$ (T | NT)*

Like a Regular Expression / DFA / NFA, a Context Free Grammar is a mathematical model

Context Free Grammars : A First Look

assign_stmt → *id := expr ;*

expr → *expr operator term*

expr → *term*

term → *id*

term → *real*

term → *integer*

operator → +

operator → -

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

Simply stated: **Grammars / production rules allow us to "rewrite" and "identify" correct syntax.**

Derivation

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

assign_stmt	*assign_stmt* → *id := expr ;*
→ *id := expr ;*	*expr* → *expr operator term*
→*id := expr operator term;*	*expr* → *expr operator term*
→*id := expr operator term operator term;*	*expr* → *term*
→ *id := term operator term operator term;*	*term* → *id*
→ *id := id operator term operator term;*	*operator* → *+*
→ *id := id + term operator term;*	*term* → *real*
→ *id := id + real operator term;*	*operator* → *-*
→ *id := id + real - term;*	*term* → *integer*
→ *id := id + real - integer;*	

Example Grammar

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

9 Production rules

To simplify / standardize notation, we offer a synopsis of terminology.

Example Grammar - Terminology

Terminals: a,b,c,+,-,punc,0,1,…,9

Non Terminals: A,B,C,S

T or NT: X,Y,Z

Strings of Terminals: u,v,…,z in T*

Strings of T / NT: α, β, γ in (T $\cup$ NT)*

Alternatives of production rules:

$A \rightarrow \alpha_1; A \rightarrow \alpha_2; \ldots; A \rightarrow \alpha_k; \Rightarrow A \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_1$

First NT on LHS of 1st production rule is designated as start symbol !

$E \rightarrow E\ A\ E \mid (\ E\ ) \mid -E \mid id$

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

Grammar Concepts

A step in a derivation is zero or one action that replaces a NT with the RHS of a production rule.

EXAMPLE: $E \Rightarrow -E$ (the $\Rightarrow$ means "derives" in one step) using the production rule: $E \rightarrow -E$

EXAMPLE: $E \Rightarrow E\ A\ E \Rightarrow E * E \Rightarrow E * (E)$

DEFINITION: $\Rightarrow$ derives in one step

$\overset{+}{\Rightarrow}$ derives in $\geq$ one step

$\overset{*}{\Rightarrow}$ derives in $\geq$ zero steps

EXAMPLES: $\alpha\ A\ \beta \Rightarrow \alpha\ \gamma\ \beta$ if $A \rightarrow \gamma$ is a production rule

$\alpha_1 \Rightarrow \alpha_2 \Rightarrow \ldots \Rightarrow \alpha_n \rightarrow \alpha_1 \overset{*}{\Rightarrow} \alpha_n$; $\alpha \overset{*}{\Rightarrow} \alpha$ for all α

If $\alpha \overset{*}{\Rightarrow} \beta$ and $\beta \rightarrow \gamma$ then $\alpha \overset{*}{\Rightarrow} \gamma$

How does this relate to Languages?

Let G be a CFG with start symbol S. Then $S \overset{+}{\Rightarrow} W$ (where W has no non-terminals) represents the language generated by G, denoted L(G). So $W \in L(G) \Leftrightarrow S \overset{+}{\Rightarrow} W$.

W : is a sentence of G

When $S \Rightarrow \alpha$ (and α may have NTs) it is called a **sentential form of G.**

EXAMPLE: id * id is a sentence

Here's the derivation:

$$E \Rightarrow E\ A\ E \Rightarrow E * E \Rightarrow id * E \Rightarrow id * id$$

Sentential forms

$$E \overset{*}{\Rightarrow} id * id$$

Other Derivation Concepts

Leftmost: **Replace the leftmost non-terminal symbol**

$$E \underset{lm}{\Rightarrow} E\ A\ E \underset{lm}{\Rightarrow} id\ A\ E \underset{lm}{\Rightarrow} id * E \underset{lm}{\Rightarrow} id * id$$

Rightmost: **Replace the leftmost non-terminal symbol**

$$E \underset{rm}{\Rightarrow} E\ A\ E \underset{rm}{\Rightarrow} E\ A\ id \underset{rm}{\Rightarrow} E * id \underset{rm}{\Rightarrow} id * id$$

Derivations: **Actions to parse input can be represented pictorially in a parse tree.**

Examples of LM / RM Derivations

$E \rightarrow E\ A\ E \mid (\ E\ ) \mid -E \mid id$

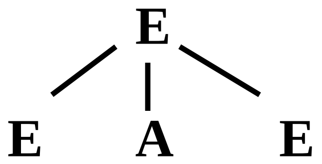
$A \rightarrow + \mid - \mid * \mid / \mid \uparrow$

A leftmost derivation of : id + id * id

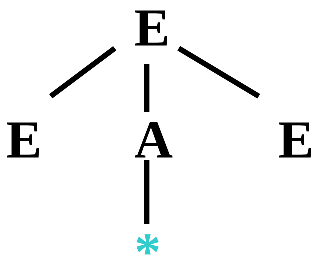
A rightmost derivation of : id + id * id

Derivations & Parse Tree

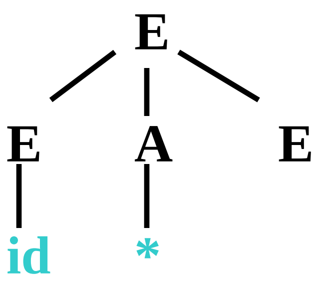
$$E \Rightarrow E\ A\ E$$



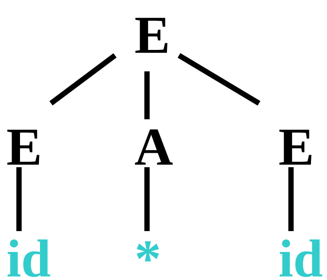
$$\Rightarrow E * E$$



$$\Rightarrow \mathrm{id} * E$$



$$\Rightarrow \mathrm{id} * \mathrm{id}$$



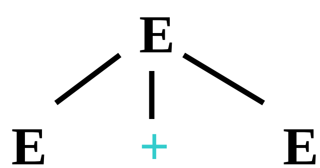
Parse Trees and Derivations

Consider the expression grammar:

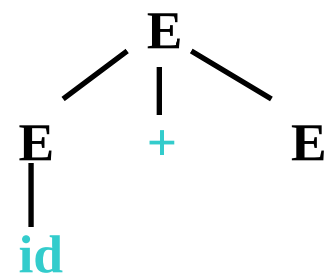
$$E \rightarrow E+E \mid E*E \mid (E) \mid -E \mid id$$

Leftmost derivations of $id + id * id$

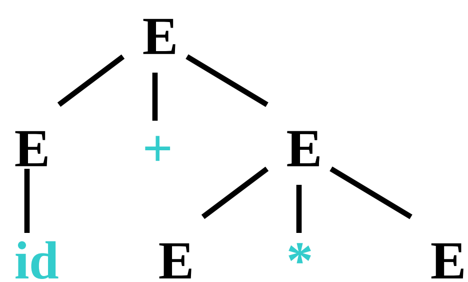
$E \Rightarrow E + E$



$E + E \Rightarrow id + E$

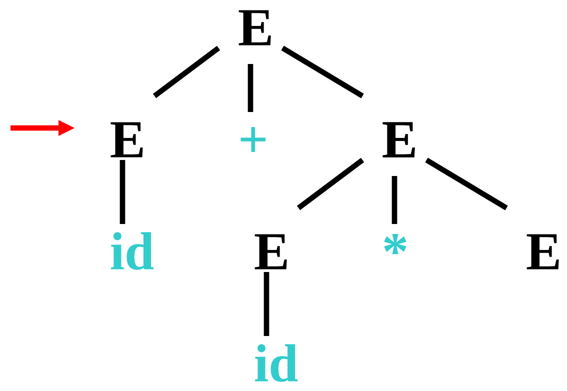


$id + E \Rightarrow id + E * E$

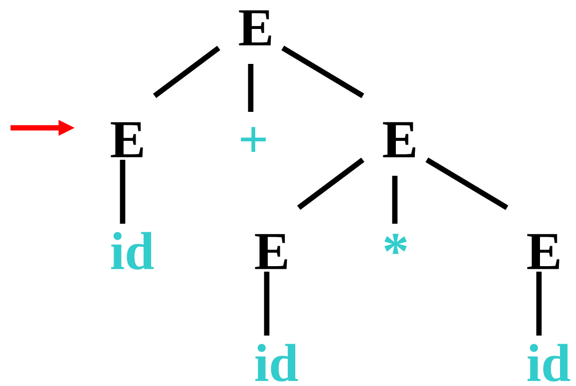


Parse Tree & Derivations - continued

$\mathbf{id} + E * E \Rightarrow \mathbf{id} + \mathbf{id} * E$



$\mathbf{id} + \mathbf{id} * E \Rightarrow \mathbf{id} + \mathbf{id} * \mathbf{id}$



Alternative Parse Tree & Derivation

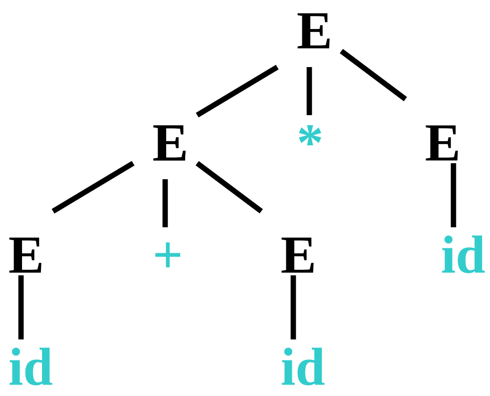
$E \Rightarrow E * E$

$\Rightarrow E + E * E$

$\Rightarrow id + E * E$

$\Rightarrow id + id * E$

$\Rightarrow id + id * id$



WHAT'S THE ISSUE HERE ?

Two distinct leftmost derivations!

Resolving Grammar Problems/Difficulties

Regular Expressions : Basis of Lexical Analysis

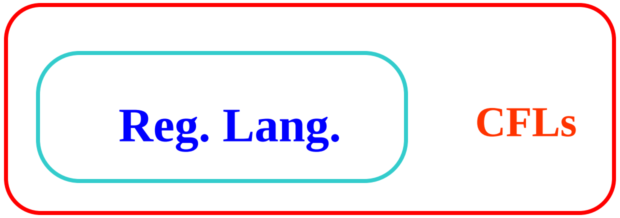
Reg. Expr. → generate/represent regular languages

Reg. Languages → smallest, most well defined class of languages

Context Free Grammars: Basis of Parsing

CFGs → represent context free languages

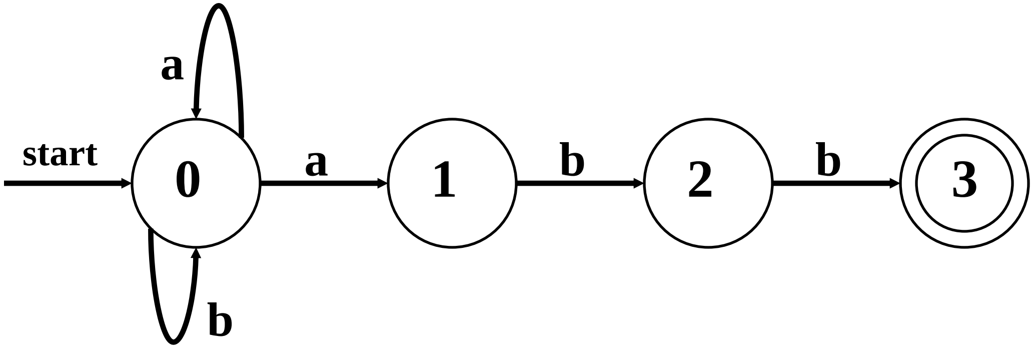
CFLs → contain more powerful languages



Resolving Problems/Difficulties – (2)

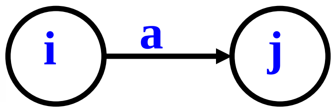
Since Reg. Lang. $\subset$ Context Free Lang., it is possible to go from reg. expr. to CFGs via NFA.

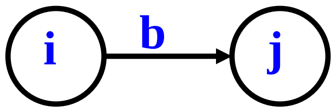
Recall: (a | b)*abb



Resolving Problems/Difficulties – (3)

Construct CFG as follows:

1. Each State I has non-terminal A_i : A_0, A_1, A_2, A_3
2. If 

then $A_i \rightarrow a\,A_j$
3. If 

then $A_i \rightarrow bA_j$
4. If I is an accepting state, $A_i \rightarrow \in$: $A_3 \rightarrow \in$
5. If I is a starting state, A_i is the start symbol : A_0

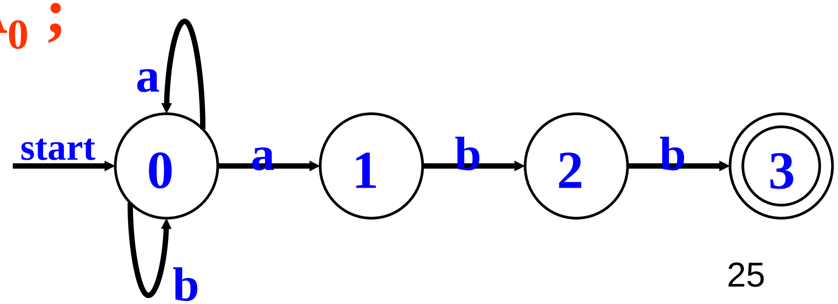
$T=\{a,b\}$, $NT=\{A_0, A_1, A_2, A_3\}$, $S = A_0$

$PR =\{ A_0 \rightarrow aA_0 \mid aA_1 \mid bA_0$;

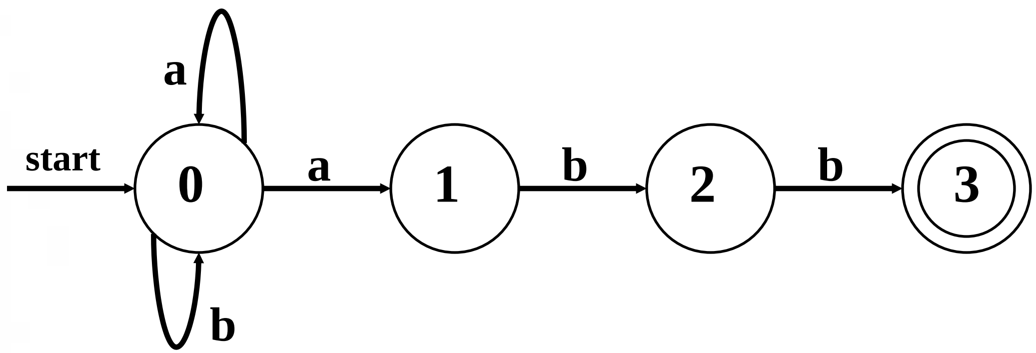
$A_1 \rightarrow bA_2$;

$A_2 \rightarrow bA_3$;

$A_3 \rightarrow \in \}$



How Does This CFG Derive Strings ?



vs.

$A_0 \rightarrow aA_0, A_0 \rightarrow aA_1$

$A_0 \rightarrow bA_0, \ A_1 \rightarrow bA_2$

$A_2 \rightarrow bA_3, A_3 \rightarrow \in$

How is abaabb derived in each ?

Regular Expressions vs. CFGs

Regular expressions for lexical syntax

1. **CFGs are overkill, lexical rules are quite simple and straightforward**
2. **REs – concise / easy to understand**
3. **More efficient lexical analyzer can be constructed**
4. **RE for lexical analysis and CFGs for parsing promotes modularity, low coupling & high cohesion.**

CFGs : Match tokens "(" ")", begin / end, if-then-else, whiles, proc/func calls, …

Intended for structural associations between tokens !

Are tokens in correct order ?

Resolving Grammar Difficulties : Motivation

1. **Humans write / develop grammars**
2. **Different parsing approaches have different needs**

Top-Down vs. Bottom-Up

- **For: 1 $\rightarrow$ remove "errors"**
- **For: 2 $\rightarrow$ put / redesign grammar**

Grammar Problems

- **- ambiguity**
- **- $\in$-moves**
- **- cycles**
- **- left recursion**
- **- left factoring**

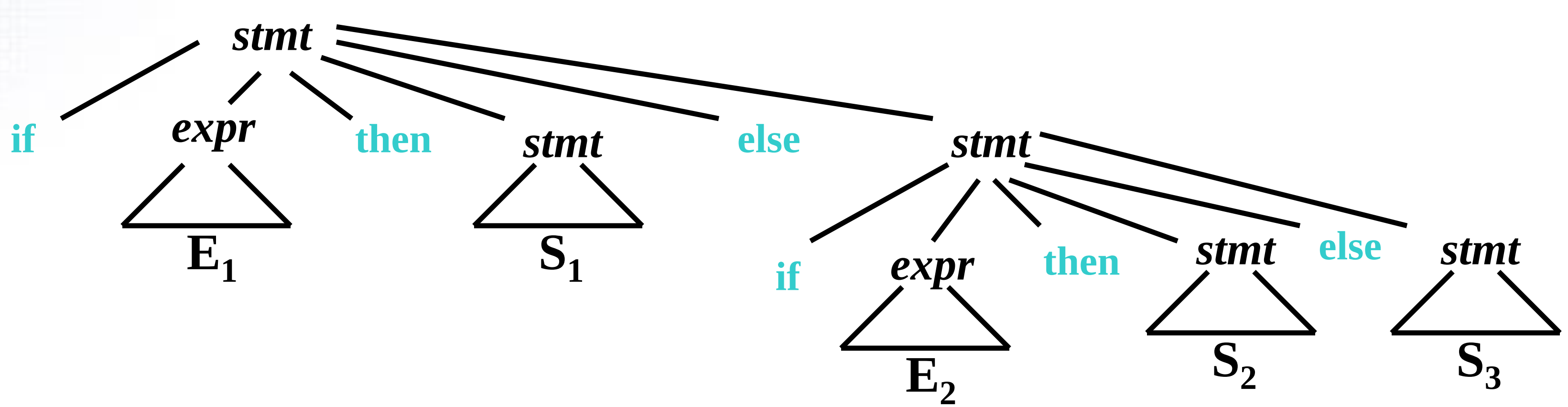
Resolving Problems: Ambiguous Grammars

Consider the following grammar segment:

$$stmt \rightarrow \textbf{if}\ expr\ \textbf{then}\ stmt$$
$$\mid\ \textbf{if}\ expr\ \textbf{then}\ stmt\ \textbf{else}\ stmt$$
$$\mid\ \textbf{other}\ \ (\text{any other statement})$$

What's problem here ?

Let's consider a simple parse tree:



Else **<u>must</u> match to previous then.**

Structure indicates parse sub-tree for expression.

Example : What Happens with this string?

If E_1 then if E_2 then S_1 else S_2

How is this parsed ?

if E_1 then
    if E_2 then
        S_1
    else
        S_2

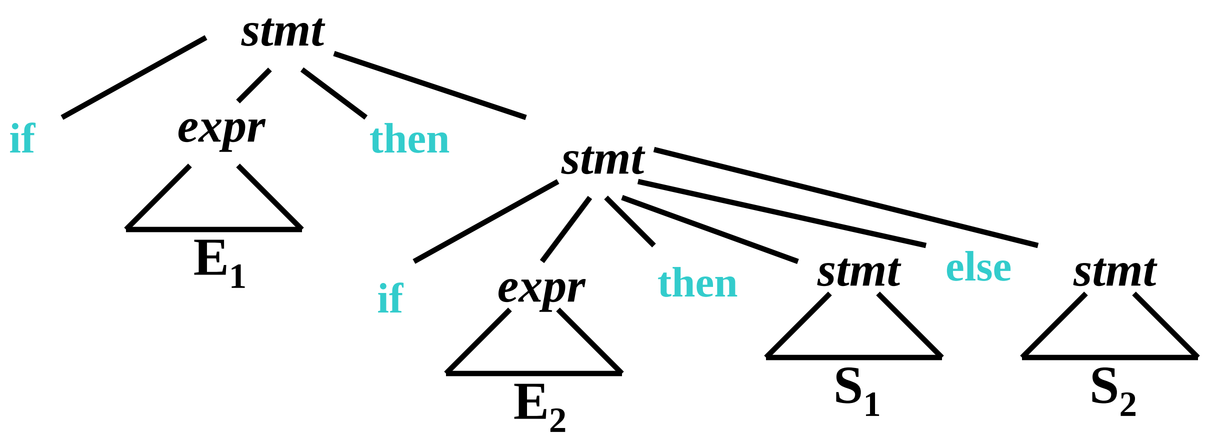
vs.

if E_1 then
    if E_2 then
        S_1
else
        S_2

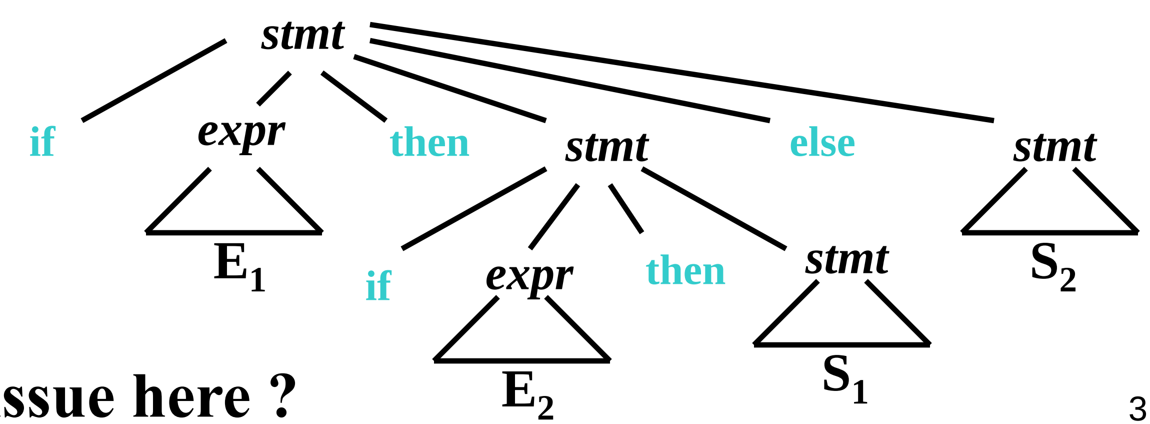
What's the issue here ?

Parse Trees for Example

Form 1:



Form 2:



What's the issue here ?

Removing Ambiguity

Take Original Grammar:

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

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

Revise to remove ambiguity:

stmt → *matched_stmt* | *unmatched_stmt*

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

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

Resolving Difficulties : Left Recursion

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

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

$A \Rightarrow A\alpha \Rightarrow A\alpha\alpha \Rightarrow A\alpha\alpha\alpha$ … etc. $A \rightarrow A\alpha \mid \beta$

Take left recursive grammar:

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

To the following:

$$A \rightarrow \beta A'$$

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

Why is Left Recursion a Problem ?

Consider:

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

Derive : $id + id + id$

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

How can left recursion be removed ?

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

What does this generate?

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

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

…

How does this build strings ?

What does each string have to start with ?

Resolving Difficulties : Left Recursion (2)

Informal Discussion:

Take all productions for <u>A</u> and order as:

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

Where no β_i begins with A.

Now apply concepts of previous slide:

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

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

For our example:

$$E \rightarrow E + T \mid T \longrightarrow \begin{cases} E \rightarrow TE' \\ E' \rightarrow + TE' \mid \in \end{cases}$$

$$T \rightarrow T * F \mid F \longrightarrow \begin{cases} T \rightarrow FT' \\ T' \rightarrow * FT' \mid \in \end{cases}$$

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

Resolving Difficulties : Left Recursion (3)

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

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

Algorithm:

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

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

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

    `for` *j* `:= 1 to` *i* `– 1 do begin`

    replace each production of the form $A_i \to A_j\gamma$

    by the productions $A_i \to \delta_1\gamma \mid \delta_2\gamma \mid \ldots \mid \delta_k\gamma$

    where $A_j \to \delta_1|\delta_2|\ldots|\delta_k$ are all current A_j productions ;

    `end`

    eliminate the immediate left recursion among A_i productions

`end`

Removing Difficulties : $\in$-Moves

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

Why does this work ?

Examples:

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

A is Grammar $\in$-free if:

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

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

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

Removing Difficulties : Left Factoring

Problem : Uncertain which of 2 rules to choose:

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

When do you know which one is valid ?

What's the general form of *stmt* ?

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

α : **if** *expr* **then** *stmt*

β_1: **else** *stmt* β_2 : ϵ

Transform to:

$A \rightarrow \alpha\ A'$

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

EXAMPLE:

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

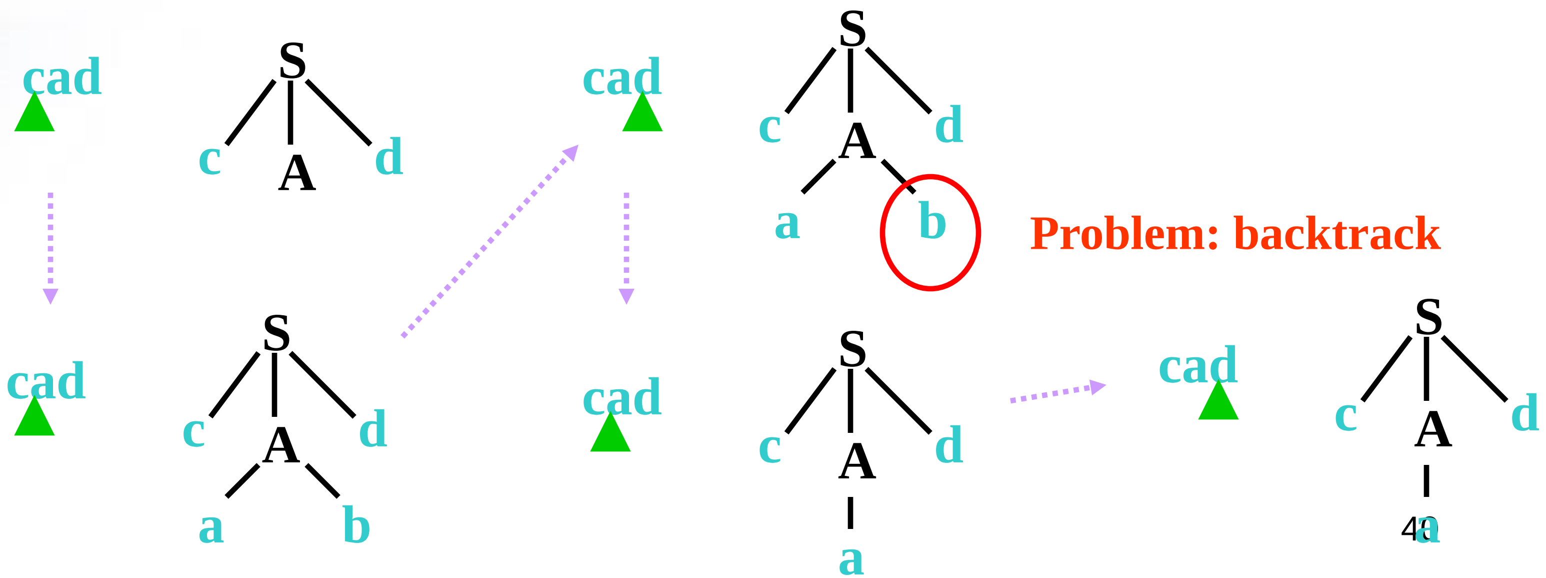
$rest \rightarrow \textbf{else}\ stmt \mid \epsilon$

Top-Down Parsing

1. Can be viewed as an attempt to find a leftmost derivation for an input string.
2. Why ?
   1. By always replacing the leftmost non-terminal symbol via a production rule, we are guaranteed of developing a parse tree in a left-to-right fashion that is consistent with scanning the input.
   2. $A \Rightarrow aBc \Rightarrow adDc \Rightarrow adec$ (scan a, scan d, scan e, scan c - accept!)
3. Recursive-descent parsing concepts – may involve backtracking
4. Predictive parsing
   1. Recursive / Brute force technique
   2. non-recursive / table driven
5. Error recovery
6. Implementation

Recursive Descent Parsing Concepts

- **General category of Parsing Top-Down**
- **Choose production rule based on input symbol**
- **May require backtracking to correct a wrong choice.**
- **Example:** $S \rightarrow c\,A\,d$, $A \rightarrow ab \mid a$ **input: cad**



Predictive Parsing : Recursive

1. To eliminate backtracking, what must we do/be sure of for grammar?
   1. no left recursion
   2. apply left factoring
2. Frequently, when grammar satisfies above conditions: current input symbol in conjunction with current non-terminal *uniquely determines* the production that needs to be applied.
3. Utilize transition diagrams:

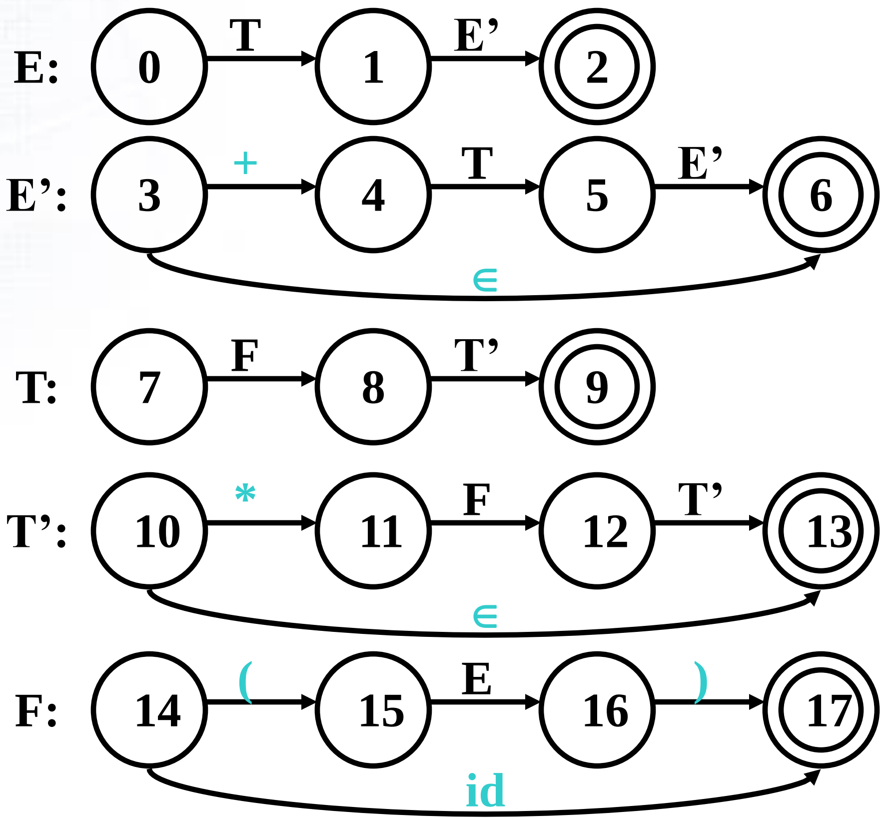
   For each non-terminal of the grammar do following:
   1. Create an initial and final state
   2. If $A \rightarrow X_1X_2 \ldots X_n$ is a production, add path with edges X_1, X_2, … , X_n
4. Once transition diagrams have been developed, apply a straightforward technique to algorithmic transition diagrams with procedure and possible recursion.

Transition Diagrams

- Unlike lexical equivalents, each edge represents a token
- Transition implies: if token, match input else <u>call proc</u>
- Recall earlier grammar and its associated transition diagrams

$E \rightarrow TE'$ $\quad$ $T \rightarrow FT'$ $\quad$ $F \rightarrow (\ E\ ) \mid id$

$E' \rightarrow +\ TE' \mid \in$ $\quad$ $T' \rightarrow *\ FT' \mid \in$

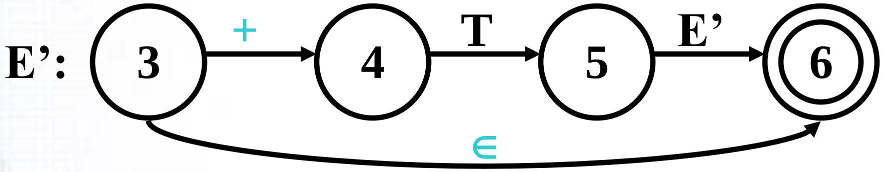


How are transition diagrams used ?

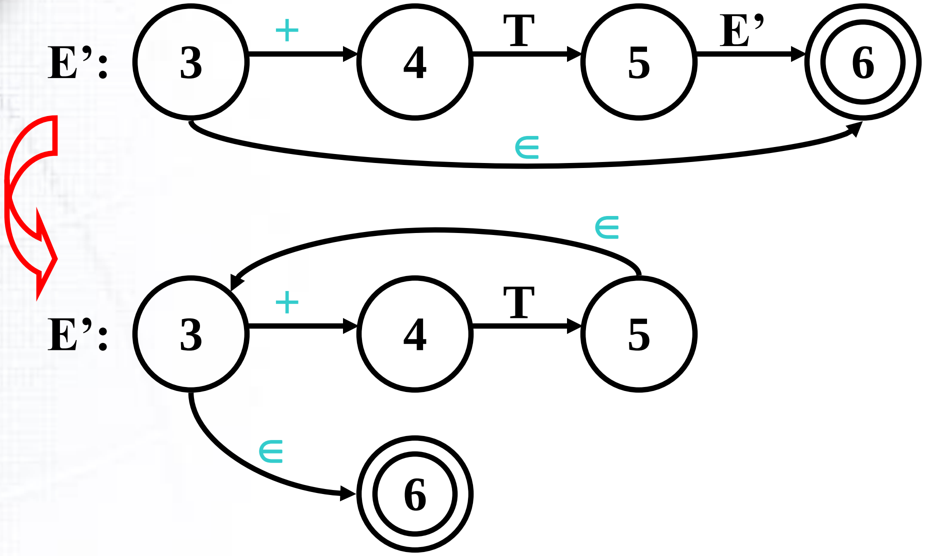
Can we simplify transition diagrams ?

Why is simplification critical ?

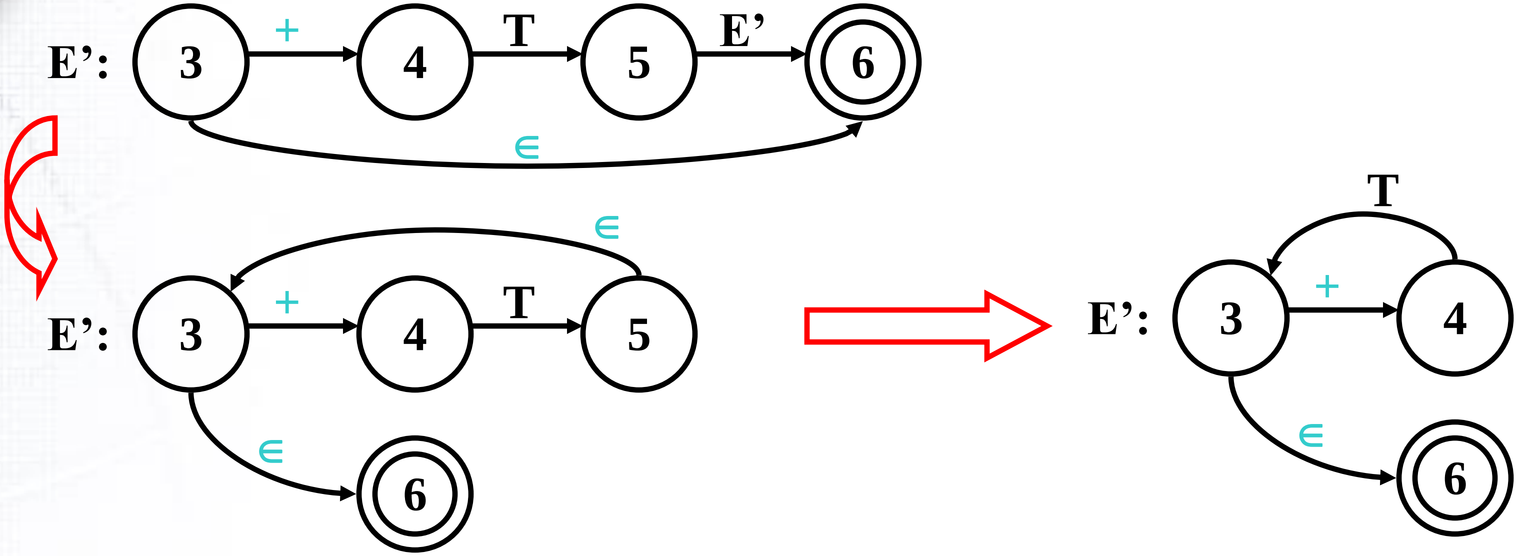
How can Transition Diagrams be Simplified ?



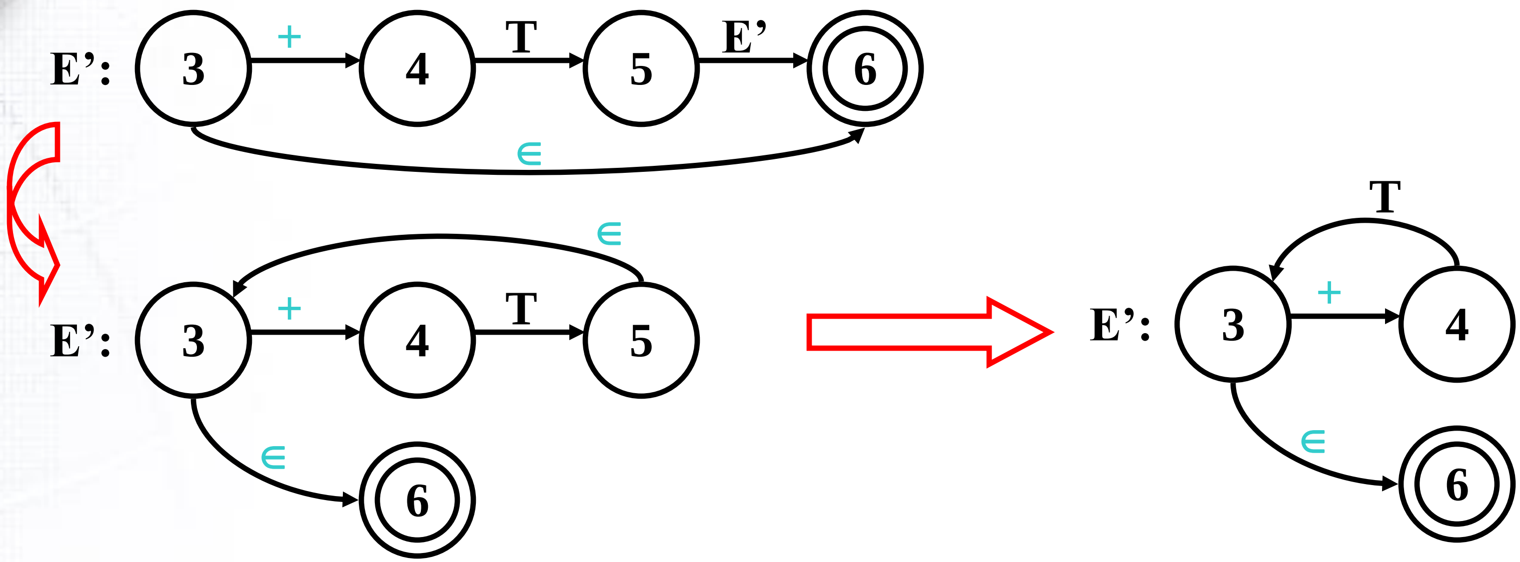
How can Transition Diagrams be Simplified ? (2)

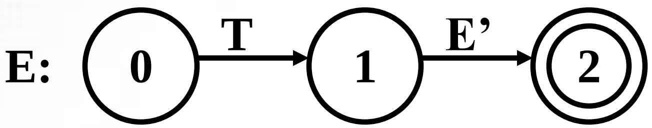


How can Transition Diagrams be Simplified ? (3)

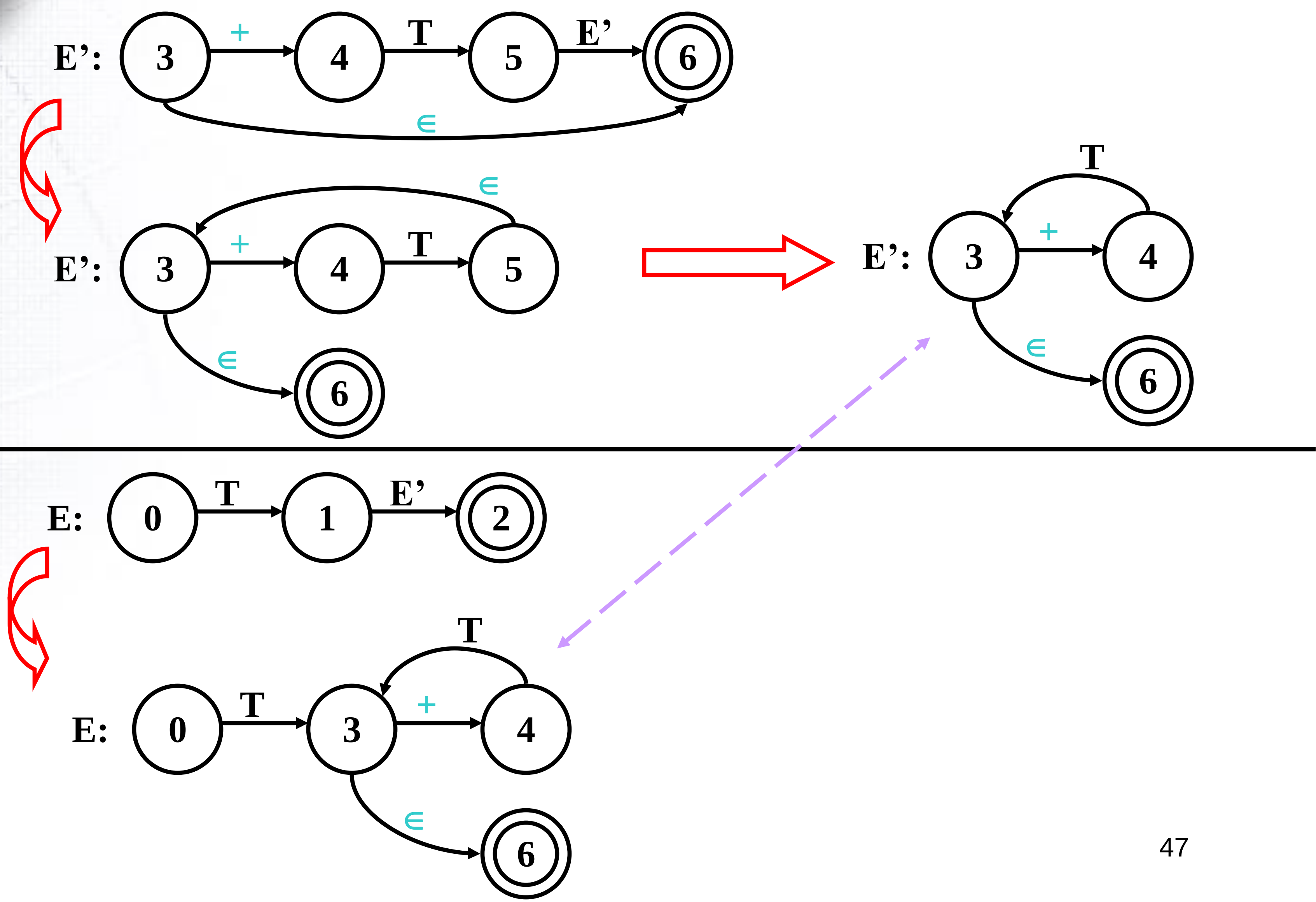


How can Transition Diagrams be Simplified ? (4)



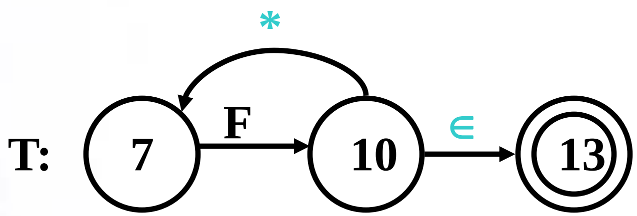


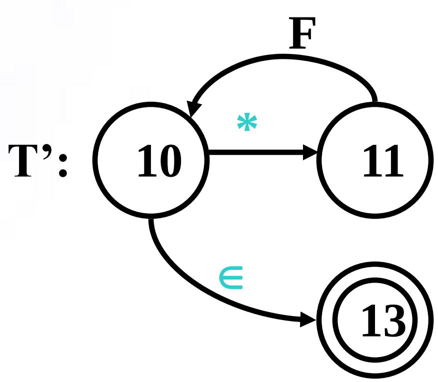
How can Transition Diagrams be Simplified ? (5)

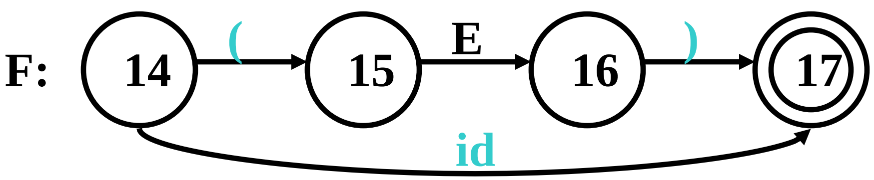


Additional Transition Diagram Simplifications

- Similar steps for T and T’
- Simplified Transition diagrams:







Motivating Table-Driven Parsing

1. Left to right scan input

2. Find leftmost derivation

Terminator

Grammar: $E \rightarrow TE'$

$E' \rightarrow +TE' \mid \in$

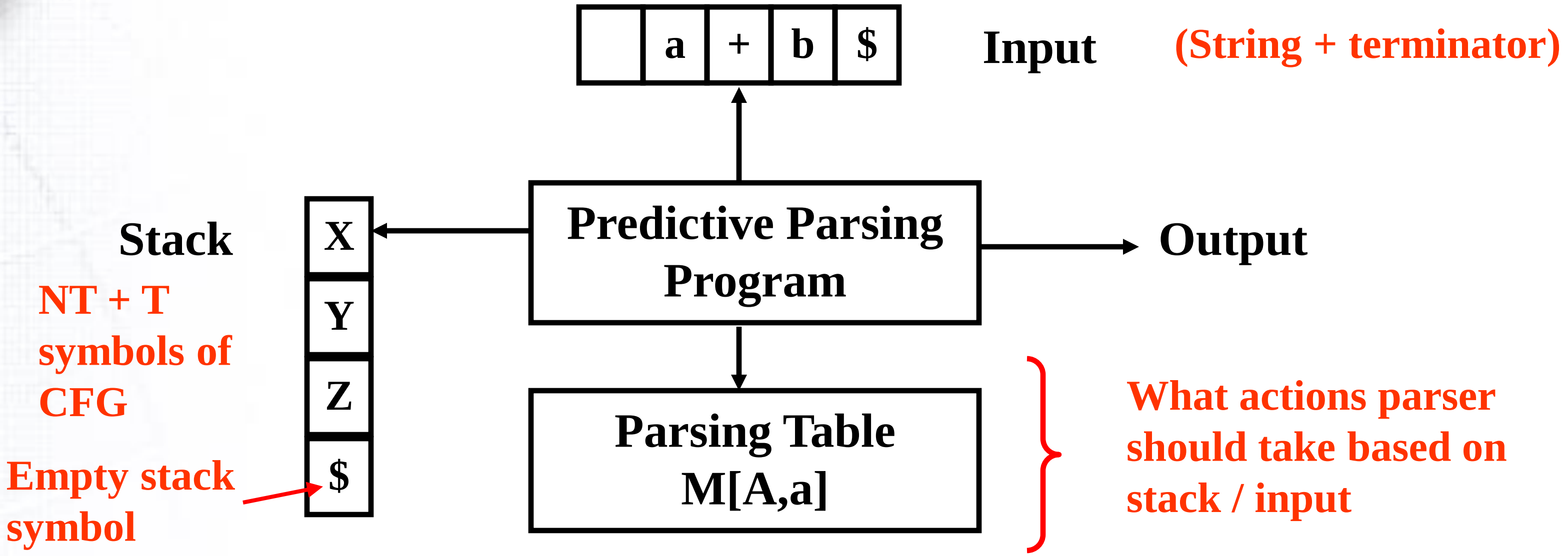
$T \rightarrow id$

Input : id + id $

Derivation: $E \Rightarrow$

Processing Stack:

Non-Recursive / Table Driven



General parser behavior: X : top of stack a : current input

1. When X=a = $ halt, accept, success
2. When X=a ≠ $, POP X off stack, advance input, go to 1.
3. When X is a non-terminal, examine M[X,a]

   if it is an error → call recovery routine

   if M[X,a] = {X → UVW}, POP X, PUSH W,V,U

   DO NOT expend any input

Algorithm for Non-Recursive Parsing

Set *ip* to point to the first symbol of w$;

repeat

let X be the top stack symbol and *a* the symbol pointed to by *ip*;

if X is terminal or $ **then**

if X=a **then**

pop X from the stack and advance *ip*

else *error()*

else /* X is a non-terminal */

if $M[X,a] = X \rightarrow Y_1 Y_2 \ldots Y_k$ **then begin**

pop X from stack;

push $Y_k, Y_{k-1}, \ldots, Y_1$ onto stack, with Y_1 on top

output the production $X \rightarrow Y_1 Y_2 \ldots Y_k$

end

else *error()*

until X=$ /* stack is empty */

Input pointer

May also execute other code based on the production used

Example

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

Our well-worn example !

Table M

Non-terminal	INPUT SYMBOL					
	id	+	*	(	)	$
E	E→TE’			E→TE’		
E’		E’→+TE’			E’→∈	E’→∈
T	T→FT’			T→FT’		
T’		T’→∈	T’→*FT’		T’→∈	T’→∈
F	F→id			F→(E)		

Trace of Example

STACK	INPUT	OUTPUT

Trace of Example

STACK	INPUT	OUTPUT
$E	id + id * id$	
$E'T	id + id * id$	$E \rightarrow TE'$
$E'T'F	id + id * id$	$T \rightarrow FT'$
$E'T'id	id + id * id$	$F \rightarrow id$
$E'T'	+ id * id$	
$E'	+ id * id$	$T' \rightarrow \in$
$E'T+	+ id * id$	$E' \rightarrow +TE'$
$E'T	id * id$	
$E'T'F	id * id$	$T \rightarrow FT'$
$E'T'id	id * id$	$F \rightarrow id$
$E'T'	* id$	
$E'T'F*	* id$	$T' \rightarrow *FT'$
$E'T'F	id$	
$E'T'id	id$	$F \rightarrow id$
$E'T'	$	
$E'	$	$T' \rightarrow \in$
$	$	$E' \rightarrow \in$

Expend Input

Leftmost Derivation for the Example

The leftmost derivation for the example is as follows:

$$E \Rightarrow TE' \Rightarrow FT'E' \Rightarrow \mathbf{id}\ T'E' \Rightarrow \mathbf{id}\ E' \Rightarrow \mathbf{id} + TE' \Rightarrow \mathbf{id} + FT'E'$$

$$\Rightarrow \mathbf{id} + \mathbf{id}\ T'E' \Rightarrow \mathbf{id} + \mathbf{id} * FT'E' \Rightarrow \mathbf{id} + \mathbf{id} * \mathbf{id}\ T'E'$$

$$\Rightarrow \mathbf{id} + \mathbf{id} * \mathbf{id}\ E' \Rightarrow \mathbf{id} + \mathbf{id} * \mathbf{id}$$

What's the Missing Puzzle Piece ?

Constructing the Parsing Table M !

1st : Calculate First & Follow for Grammar

2nd: Apply Construction Algorithm for Parsing Table

(We'll see this shortly)

Basic Tools:

First: **Let α be a string of grammar symbols. First(α) is the set that includes every terminal that appears leftmost in α or in any string originating from α.**
NOTE: If $\alpha \overset{*}{\Rightarrow} \epsilon$, then ϵ is First(α).

Follow: **Let A be a non-terminal. Follow(A) is the set of terminals a that can appear directly to the right of A in some sentential form.**

($S \overset{*}{\Rightarrow} \alpha A a \beta$, for some α and β).

LL(1) Grammars

L : Scan input from Left to Right

L : Construct a Leftmost Derivation

1 : Use "1" input symbol as lookahead in conjunction with stack to decide on the parsing action

LL(1) grammars == they have no multiply-defined entries in the parsing table.

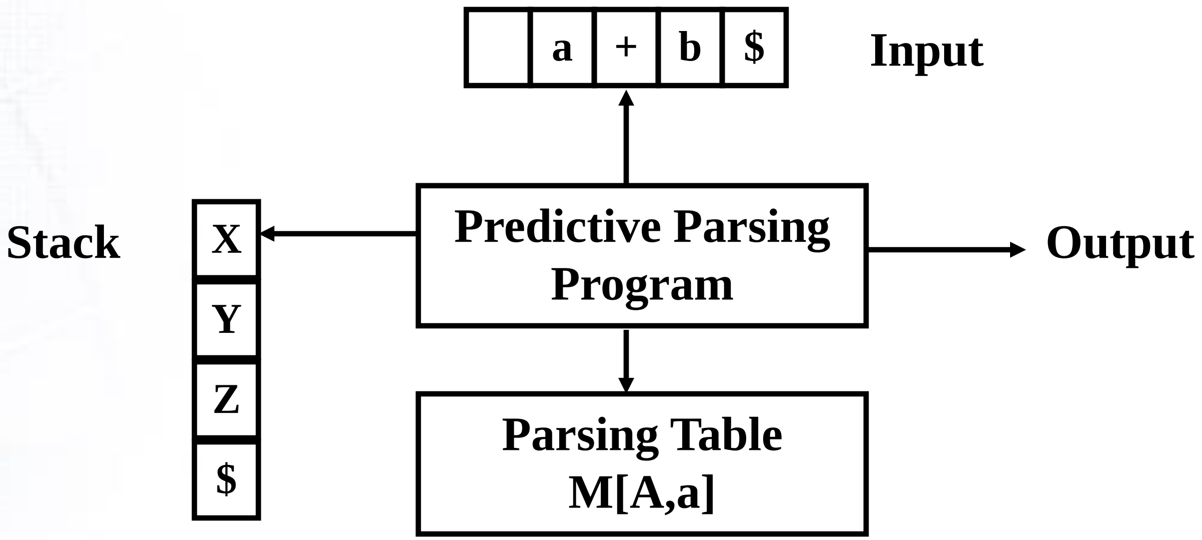
Properties of LL(1) grammars:

1. Grammar can't be ambiguous or left recursive
2. Grammar is LL(1) $\Leftrightarrow$ when $A \rightarrow \alpha \mid \beta$
   a. α & β do not derive strings starting with the same terminal a
   b. Either α or β can derive $\in$, but not both.

Note: It may not be possible for a grammar to be manipulated into an LL(1) grammar

Error Recovery

When Do Errors Occur? Recall Predictive Parser Function:



1. **If X is a terminal and it doesn't match input.**
2. **If M[X, Input] is empty – No allowable actions**

Consider two recovery techniques:

A. **Panic Mode**

B. **Phrase-level Recovery**

Panic-Mode Recovery

Assume a non-terminal on the top of the stack.

1. Idea:

   skip symbols on the input until a token in a selected set of *synchronizing* tokens is found.

2. The choice for a synchronizing set is important.

   Some ideas:

   a. Define the synchronizing set of A to be FOLLOW(A). then skip input until a token in FOLLOW(A) appears and then pop A from the stack. Resume parsing...

   b. Add symbols of FIRST(A) into synchronizing set. In this case we skip input and once we find a token in FIRST(A) we resume parsing from A.

   c. Productions that lead to $\in$ if available might be used.

3. **If a terminal appears on top of the stack and does not match to the input == pop it and and continue parsing (issuing an error message saying that the terminal was inserted).**

Panic Mode Recovery, II

General Approach: Modify the empty cells of the Parsing Table.

1. **if M[A,a] = {empty} and a belongs to Follow(A) then we set M[A,a] = "synch" (indicate synchronizing token obtained from the FOLLOW set of the nonterminal in question)**

Error-recovery Strategy :

If A=top-of-the-stack and a=current-input,

1. **If A is NT and M[A,a] = {empty} then skip a from the input.**
2. **If A is NT and M[A,a] = {synch} then pop A.**
3. **If A is a terminal and A!=a then pop token (essentially inserting it).**

Phrase-Level Recovery

1. **Fill in blanks entries of parsing table with error handling routines**
2. **These routines**
   a) **Modify stack and / or input stream**
   b) **Issue error message**
3. **Problems:**
   a) **Modifying stack has to be done with care, so as to not create possibility of derivations that aren't in language**
   b) **Infinite loops must be avoided**
4. **Can be used in conjunction with panic mode to have more complete error handling**

Final Comments – Top-Down Parsing

So far,

1. **We've examined grammars and language theory and its relationship to parsing**
2. **Key concepts: Rewriting grammar into an acceptable form**
3. **Examined Top-Down parsing:**
   a) **Brute Force : Transition diagrams & recursion**
   b) **Elegant : Table driven**
4. **We've identified its shortcomings:**

   Not all grammars can be made LL(1) !

5. **Bottom-Up Parsing - Future**

Bottom-up Parsing

Parsing Techniques

Top-down parsers ***(LL(1), recursive descent)***

- Start at the root of the parse tree from the start symbol and grow toward leaves (similar to a derivation)
- Pick a production and try to match the input
- Bad "pick" $\Rightarrow$ may need to backtrack
- Some grammars are backtrack-free *(predictive parsing)*

Bottom-up parsers ***(LR(1), operator precedence)***

Start at the leaves and grow toward root

We can think of the process as reducing the input string to the start symbol

At each reduction step a particular substring matching the right-side of a production is replaced by the symbol on the left-side of the production

Bottom-up parsers handle a large class of grammars

Bottom-up Parsing

A general style of bottom-up syntax analysis, known as shift-reduce parsing.

Two types of bottom-up parsing:

1. Operator-Precedence parsing
2. LR parsing

Bottom Up Parsing

- "Shift-Reduce" Parsing
- Reduce a string to the start symbol of the grammar.
- At every step a particular sub-string is matched (in left-to-right fashion) to the right side of some production and then it is substituted by the non-terminal in the left hand side of the production.

Consider:

$$S \rightarrow aABe$$
$$A \rightarrow Abc \mid b$$
$$B \rightarrow d$$

abbcde
aAbcde
aAde
aABe
S

Reverse order

Rightmost Derivation:

$$S \Rightarrow aABe \Rightarrow aAde \Rightarrow aAbcde \Rightarrow abbcde$$

Handles

- Handle of a string: Substring that matches the RHS of some production AND whose reduction to the non-terminal on the LHS is a step along the reverse of some rightmost derivation.
- **Formally:**
  - **handle of a right sentential form γ is <A $\rightarrow$ β, location of β in γ> ,** *that satisfies the above property.*
  - i.e. A $\rightarrow$ β is a handle of $\alpha\beta\gamma$ at the location immediately after the end of α, if:

$$S \overset{rm}{\underset{*}{\Rightarrow}} \alpha A \gamma \overset{rm}{\Rightarrow} \alpha\beta\gamma$$

- A certain sentential form may have many different handles.
- Right sentential forms of a non-ambiguous grammar have one *unique* handle

Example

Consider:

$$S \rightarrow aABe$$
$$A \rightarrow Abc \mid b$$
$$B \rightarrow d$$

$$S \Rightarrow \underline{aABe} \Rightarrow aA\underline{d}e \Rightarrow a\underline{Abc}de \Rightarrow a\underline{b}bcde$$

It follows that:

$S \rightarrow aABe$ is a handle of $\underline{aABe}$ in location 1.

$B \rightarrow d$ is a handle of $aA\underline{d}e$ in location 3.

$A \rightarrow Abc$ is a handle of $a\underline{Abc}de$ in location 2.

$A \rightarrow b$ is a handle of $a\underline{b}bcde$ in location 2.

Handle Pruning

- A rightmost derivation in reverse can be obtained by “handle-pruning.”
- Apply this to the previous example.

$S \rightarrow aABe$

$A \rightarrow Abc \mid b$

$B \rightarrow d$

abbcde

Find the handle = b at loc. 2

aAbcde

b at loc. 3 is not a handle:

aAAcde

... blocked.

Also Consider:

$E \rightarrow E + E \mid E * E \mid$

$\mid (E) \mid id$

Derive id+id*id
By two different Rightmost derivations

Handle-pruning, Bottom-up Parsers

The process of discovering a handle & reducing it to the appropriate left-hand side is called *handle pruning*.

Handle pruning forms the basis for a bottom-up parsing method.

Rightmost Derivation:

S $\Rightarrow$ **aABe** $\Rightarrow$ **aAde** $\Rightarrow$ **aAbcde** $\Rightarrow$ **abbcde**

To construct a rightmost derivation

$$S = \gamma_0 \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \ldots \Rightarrow \gamma_{n-1} \Rightarrow \gamma_n = w$$

Apply the following simple algorithm

for i $\leftarrow$ *n to 1 by -1*

Find the handle $A_i \rightarrow \beta_i$ in γ_i

Replace β_i *with* A_i *to generate* γ_{i-1}

Shift Reduce Parsing with a Stack

- Two problems:
  - locate a handle and
  - decide which production to use (if there are more than two candidate productions).
- General Construction: using a stack:
  - "shift" input symbols into the stack until a handle is found on top of it.
  - "reduce" the handle to the corresponding non-terminal.

  - other operations:
    - "accept" when the input is consumed and only the start symbol is on the stack, also: "error"

Example

STACK	INPUT	Action
$	id + id * id$	Shift
$ id	+ id * id$	Reduce by E $\rightarrow$ id
$E	+ id * id$	

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

Example

	Stack	Input	Action
(1)	$\$$	$\mathbf{id}_1 + \mathbf{id}_2 * \mathbf{id}_3\$$	shift
(2)	$\$\mathbf{id}_1$	$+ \mathbf{id}_2 * \mathbf{id}_3\$$	reduce by $E \to \mathbf{id}$
(3)	$\$E$	$+ \mathbf{id}_2 * \mathbf{id}_3\$$	shift
(4)	$\$E +$	$\mathbf{id}_2 * \mathbf{id}_3\$$	shift
(5)	$\$E + \mathbf{id}_2$	$* \mathbf{id}_3\$$	reduce by $E \to \mathbf{id}$
(6)	$\$E + E$	$* \mathbf{id}_3\$$	shift
(7)	$\$E + E *$	$\mathbf{id}_3\$$	shift
(8)	$\$E + E * \mathbf{id}_3$	$\$$	reduce by $E \to \mathbf{id}$
(9)	$\$E + E * E$	$\$$	reduce by $E \to E * E$
(10)	$\$E + E$	$\$$	reduce by $E \to E + E$
(11)	$\$E$	$\$$	accept

More on Shift-Reduce Parsing

Viable prefixes:

The set of prefixes of a right sentential form that can appear on the stack of a Shift-Reduce parser is called Viable prefixes.

Conflicts

"shift/reduce" or "reduce/reduce"

Example:

$$
\begin{aligned}
\mathit{stmt} \rightarrow\ & \textbf{if}\ \mathit{expr}\ \textbf{then}\ \mathit{stmt} \\
|\ & \textbf{if}\ \mathit{expr}\ \textbf{then}\ \mathit{stmt}\ \textbf{else}\ \mathit{stmt} \\
|\ & \textbf{other}\ \ \textbf{(any other statement)}
\end{aligned}
$$

We can't tell whether it is a handle

Stack	**Input**
if … then stmt	**else …**

Shift/ Reduce Conflict

Shift-reduce Parsing

Shift reduce parsers are easily built and easily understood

A shift-reduce parser has just four actions

- *Shift* — next word is shifted onto the stack
- *Reduce* — right end of handle is at top of stack
  - Locate left end of handle within the stack
  - Pop handle off stack & push appropriate *lhs*
- *Accept* — stop parsing & report success
- *Error* — call an error reporting/recovery routine

Handle finding is key

- **handle is on stack**
- **finite set of handles**

$\Rightarrow$ **use a DFA !**

Accept & *Error* are simple

Shift is just a push and a call to the scanner

Reduce takes |*rhs*| pops & 1 push

If handle-finding requires state, put it in the stack

Operator-Precedence Parser

- **Operator grammar**
  - small, but an important class of grammars
  - we may have an efficient operator precedence parser (a shift-reduce parser) for an operator grammar.
- In an *operator grammar*, no production rule can have:
  - ε at the right side
  - two adjacent non-terminals at the right side.
- Ex:

$E \rightarrow AB$	$E \rightarrow EOE$	$E \rightarrow E+E \mid$
$A \rightarrow a$	$E \rightarrow id$	$E*E \mid$
$B \rightarrow b$	$O \rightarrow + \mid * \mid /$	$E/E \mid id$
not operator grammar	not operator grammar	operator grammar

Precedence Relations

- In operator-precedence parsing, we define three disjoint precedence relations between certain pairs of terminals.

a <· b	b has higher precedence than a
a =· b	b has same precedence as a
a ·> b	b has lower precedence than a

- The determination of correct precedence relations between terminals are based on the traditional notions of associativity and precedence of operators. (Unary minus causes a problem).

Using Operator-Precedence Relations

- The intention of the precedence relations is to find the handle of a right-sentential form,

  <· with marking the left end,

  =· appearing in the interior of the handle, and

  ·> marking the right hand.

- In our input string $\$a_1a_2...a_n\$$, we insert the precedence relation between the pairs of terminals (the precedence relation holds between the terminals in that pair).

Using Operator -Precedence Relations

E → E+E | E-E | E*E | E/E | E^E | (E) | -E | id

The partial operator-precedence table for this grammar

	id	+	*	$
id		·>	·>	·>
+	<·	·>	<·	·>
*	<·	·>	·>	·>
$	<·	<·	<·	

- Then the input string id+id*id with the precedence relations inserted will be:

  $ <· id ·> + <· id ·> * <· id ·> $

To Find The Handles

1. Scan the string from left end until the first ·> is encountered.
2. Then scan backwards (to the left) over any =· until a <· is encountered.
3. The handle contains everything to left of the first ·> and to the right of the <· is encountered.

$ <· id ·> + <· id ·> * <· id ·> $	E → id	$ id + id * id $
$ <· + <· id ·> * <· id ·> $	E → id	$ E + id * id $
$ <· + <· * <· id ·> $	E → id	$ E + E * id $
$ <· + <· * ·> $	E → E*E	$ E + E * ·E $
$ <· + ·> $	E → E+E	$ E + E $
$ $		$ E $

Operator-Precedence Parsing Algorithm

The input string is w$, the initial stack is $ and a table holds precedence relations between certain terminals

Algorithm:

set p to point to the first symbol of w$;
repeat forever
    if ($ is on top of the stack **and** p points to $) **then** **return**
    else {
        let a be the topmost terminal symbol on the stack and let b be the symbol pointed to by p;
        if ($a \lessdot b$ or $a \doteq b$) **then** {   /* SHIFT */
            push b onto the stack;
            advance p to the next input symbol;
        }
        else if ($a \gtrdot b$) **then**   /* REDUCE */
            repeat pop stack
            until (the top of stack terminal is related by $\lessdot$ to the terminal most recently popped);
        else error();
    }

Operator-Precedence Parsing Algorithm -- Example

stack	***input***	***action***		
$	id+id*id$	$ <· id	shift	
$id	+id*id$	id ·> +	reduce	E → id
$	+id*id$	shift		
$+	id*id$	shift		
$+id	*id$	id ·> *	reduce	E → id
$+	*id$	shift		
$+*	id$	shift		
$+*id	$	id ·> $	reduce	E → id
$+*	$	* ·> $	reduce	E → E*E
$+	$	+ ·> $	reduce	E → E+E
$	$	accept		

	id	+	*	$
id		·>	·>	·>
+	<·	·>	<·	·>
*	<·	·>	·>	·>
$	<·	<·	<·	

How to Create Operator-Precedence Relations

- We use associativity and precedence relations among operators.

1. If operator θ_1 has higher precedence than operator θ_2,
   ➔ $\theta_1 \gtrdot \theta_2$ and $\theta_2 \lessdot \theta_1$

2. If operator θ_1 and operator θ_2 have equal precedence,
   they are left-associative ➔ $\theta_1 \gtrdot \theta_2$ and $\theta_2 \gtrdot \theta_1$
   they are right-associative ➔ $\theta_1 \lessdot \theta_2$ and $\theta_2 \lessdot \theta_1$

3. For all operators θ, $\theta \lessdot \text{id}$, $\text{id} \gtrdot \theta$, $\theta \lessdot ($, $(\lessdot \theta$, $\theta \gtrdot)$, $) \gtrdot \theta$, $\theta \gtrdot \$$, and $\$ \lessdot \theta$

4. Also, let

$(\doteq)$	$\$ \lessdot ($	$\text{id} \gtrdot)$	$) \gtrdot \$$
$(\lessdot ($	$\$ \lessdot \text{id}$	$\text{id} \gtrdot \$$	$) \gtrdot)$
$(\lessdot \text{id}$			

Operator-Precedence Relations

	+	-	*	/	^	id	(	)	$
+	·>	·>	<·	<·	<·	<·	<·	·>	·>
-	·>	·>	<·	<·	<·	<·	<·	·>	·>
*	·>	·>	·>	·>	<·	<·	<·	·>	·>
/	·>	·>	·>	·>	<·	<·	<·	·>	·>
^	·>	·>	·>	·>	<·	<·	<·	·>	·>
id	·>	·>	·>	·>	·>			·>	·>
(	<·	<·	<·	<·	<·	<·	<·	=·	
)	·>	·>	·>	·>	·>			·>	·>
$	<·	<·	<·	<·	<·	<·	<·		

Error Recovery in Operator-Precedence Parsing

Error Cases:

1. No relation holds between the terminal on the top of stack and the next input symbol.
2. A handle is found (reduction step), but there is no production with this handle as a right side

Error Recovery:

1. Each empty entry is filled with a pointer to an error routine.
2. Decides the popped handle “looks like” which right hand side. And tries to recover from that situation.

Disadvantages of Operator Precedence Parsing

- **Disadvantages**:
  - It cannot handle the unary minus (the lexical analyzer should handle the unary minus).
  - Small class of grammars.
  - Difficult to decide which language is recognized by the grammar.

- **Advantages**:
  - simple
  - powerful enough for expressions in programming languages

Precedence Functions - Tutorial

- Compilers using operator precedence parsers do not need to store the table of precedence relations.
- The table can be encoded by two precedence functions f and g that map terminal symbols to integers.
- For symbols a and b.

$$f(a) < g(b) \quad \text{whenever} \quad a \lessdot b$$

$$f(a) = g(b) \quad \text{whenever} \quad a \doteq b$$

$$f(a) > g(b) \quad \text{whenever} \quad a \gtrdot b$$

Algorithm 4.6 Constructing precedence functions