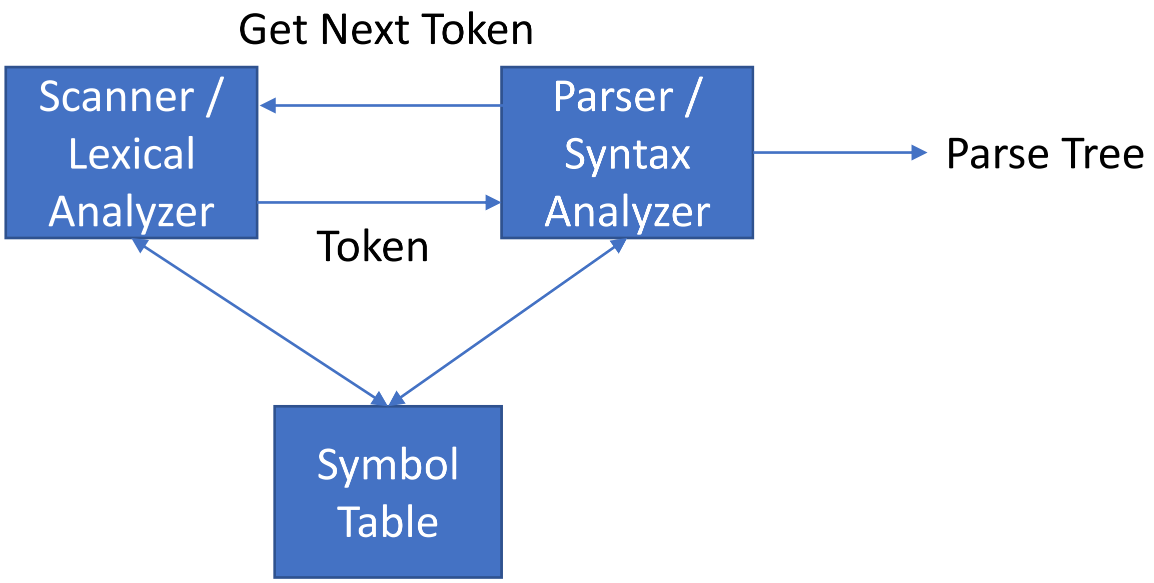
CS406: Compilers

Spring 2022

Week 4: Parsers - Top-Down Parsing (table-driven approach and background concepts), Bottom-up parsing (use of goto and action tables)

Demo

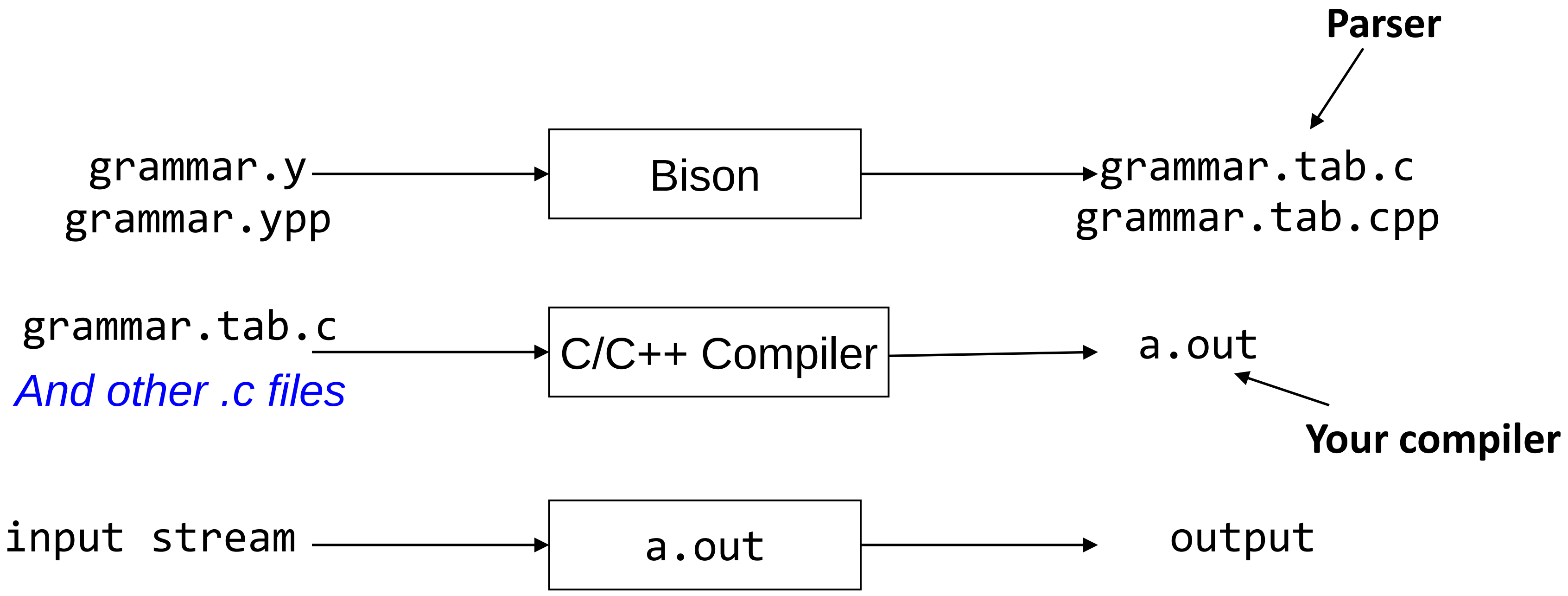
- Parser (in an implementation of a compiler)



Bison (YACC)

- Specify the grammar
- Write a lexical analyzer to process input programs and pass the tokens to parser
- Call `yyparse()` from `main`
- Write error-handlers (what happens when the compiler encounters invalid programs?)

Bison (YACC)


Parser
grammar.y
grammar.ypp
Bison
grammar.tab.c
grammar.tab.cpp
grammar.tab.c
And other .c files
C/C++ Compiler
a.out
Your compiler
input stream
a.out
output

Bison (YACC) – Input Format

```
%{
Prologue
%}
Bison declarations
%%
Grammar rules
%%
Epilogue
```

Bison (YACC) – Grammar Rules

```
%{
Prologue
%}
Bison declarations
%%
E: E PLUS E {}
 | INTEGER_LITERAL {}
 ;
%%
Epilogue
```

Bison (YACC) - Prologue

```
%{
Prologue
%}
%token PLUS INTEGER_LITERAL
%left PLUS
%%
E: E PLUS E {}
 | INTEGER_LITERAL {}
 ;
%%
Epilogue
```

Bison (YACC) - Actions

```
%{
Prologue
%}
%token PLUS INTEGER_LITERAL
%left PLUS
%%
E: E PLUS E { $$ = $1 + $3; }
 | INTEGER_LITERAL { $$ = $1; }
 ;
%%
Epilogue
```

Legal C/C++ code

Bison (YACC) – Semantic Values

```
%{
Prologue
%}
%token PLUS INTEGER_LITERAL
%left PLUS
%%          $1    $2       $3
E: E PLUS E { $$ = $1 + $3; }
 | INTEGER_LITERAL { $$ = $1; }
 ;
%%
Epilogue
```

Bison (YACC) – Helper Functions

```
%{
int yylex();
void yyerror(char *s);
%}
%token PLUS INTEGER_LITERAL
%left PLUS
%%
E: E PLUS E { $$ = $1 + $3; }
 | INTEGER_LITERAL { $$ = $1; }
 ;
%%
Epilogue
```

Bison (YACC) – Helper Functions

```
%{
#include<stdlib.h>
#include<stdio.h>
int yylex();
void yyerror(char const *s);
%}
%token PLUS INTEGER_LITERAL
%left PLUS
%%
E: E PLUS E { $$ = $1 + $3; }
 | INTEGER_LITERAL { $$ = $1; };
%%
void yyerror(char const* s) {
      fprintf(stderr,”%s\n”,s);
      exit(1);
}
```

Bison (YACC) – Integrating

- Recall that terminals are tokens
- Lexer produces tokens
  - How do the parser and lexer have a common understanding of tokens?
  - How should the Lexer return tokens?

```
//grammar.y file
…
%token PLUS INTEGER_LITERAL
%%
E: E PLUS E { $$ = $1 + $3; }
 | INTEGER_LITERAL { $$ = $1; };
%%
…
```

```
bison –d grammar.y
```

`grammar.tab.h`

```
//scanner.l file
#include"grammar.tab.h"
extern YYSTYPE yylval
%%
\+       {return PLUS;}
[0-9]+   { yylval=atoi(yytext);
         return INTEGER_LITERAL;}
.|\n     {}
%%
…
```

Bison(YACC) - More..

- %union
- %define
- error

- [Reference: Top (Bison 3.8.1) (gnu.org)]

Top-down Parsing

- Idea: we know sentence has to start with initial symbol
- Build up partial derivations by *predicting* what rules are used to expand non-terminals
  - Often called *predictive parsers*
- If partial derivation has terminal characters, *match* them from the input stream

Top-down Parsing

- Also called recursive-descent parsing
- Equivalent to finding the left-derivation for an input string
  - Recall: expand the leftmost non-terminal in a parse tree
  - Expand the parse tree in pre-order i.e., identify parent nodes before children

Top-down Parsing

↑: next symbol to be read

```
1: S -> cAd
2: A -> ab
3:     | a
```

Step	Input string	Parse tree
1	cad ↑	S

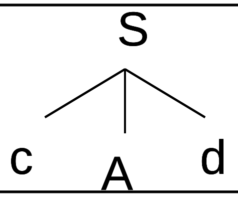
String: **cad**

Start with S

Top-down Parsing

↑: next symbol to be read

```
1: S -> cAd
2: A -> ab
3:      | a
```

Step	Input string	Parse tree
1	cad ↑	S
2	cad ↑	

String: **cad**

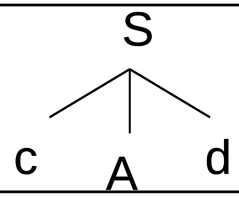
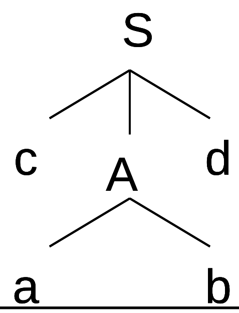
Predict rule 1

Top-down Parsing

↑: next symbol to be read

```
1: S -> cAd
2: A -> ab
3:      | a
```

String: **cad**

Step	Input string	Parse tree
1	cad ↑	S
2	cad ↑	
3	cad ↑	

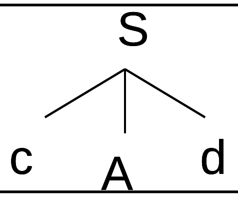
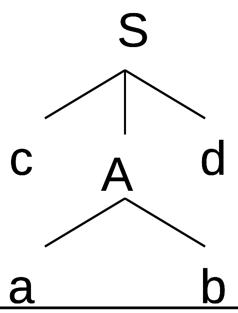
Predict rule 2

Top-down Parsing

↑: next symbol to be read

```
1:  S  ->  cAd
2:  A  ->  ab
3:        |  a
```

String: **cad**

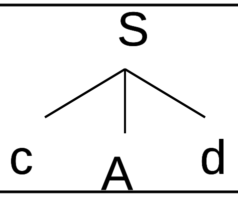
Step	Input string	Parse tree
1	$\underset{\uparrow}{c}ad$	S
2	$c\underset{\uparrow}{a}d$	
3	$ca\underset{\uparrow}{d}$	

No more non terminals!
String doesn't match.
Backtrack.

Top-down Parsing

↑: next symbol to be read

```
1: S -> cAd
2: A -> ab
3:      | a
```

Step	Input string	Parse tree
1	cad ↑	S
2	cad ↑	

String: **cad**

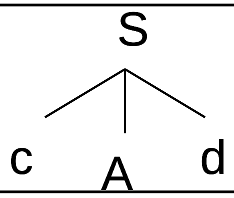
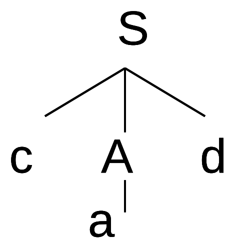
Top-down Parsing

↑: next symbol to be read

```
1:  S  ->  cAd
2:  A  ->  ab
3:         |  a
```

String: **cad**

Predict rule 3

Step	Input string	Parse tree
1	$\underset{\uparrow}{c}ad$	S
2	$c\underset{\uparrow}{a}d$	
4	$ca\underset{\uparrow}{d}$	

Top-down Parsing – Table-driven Approach

```
1: S -> F
2: S -> (S + F)
3: F -> a
```

string: `(a+a)`

string’: `(a+a)$`

	(	)	a	+	$
S	2	-	1	-	-
F	-	-	3	-	-

Assume that the table is given.

Top-down Parsing – Table-driven Approach

```
1: S -> F
2: S -> (S + F)
3: F -> a
```

string: (a+a)

string': (a+a)$

	(	)	a	+	$
S	2	-	1	-	-
F	-	-	3	-	-

Assume that the table is given.

- Table-driven (Parse Table) approach doesn't require backtracking

But how do we construct such a table?

Important Concepts: First Sets and Follow Sets

First and follow sets

- First(α): the set of terminals (and/or λ) that begin all strings that can be derived from α
  - First(A) = {x, y, λ}
  - First(xaA) = {x}
  - First (AB) = {x, y, b}
- Follow(A): the set of terminals (and/or $, but no λs) that can appear immediately after A in some partial derivation
  - Follow(A) = {b}

$S \rightarrow A\ B\ \$$

$A \rightarrow x\ a\ A$

$A \rightarrow y\ a\ A$

$A \rightarrow \lambda$

$B \rightarrow b$

First and follow sets

- $\text{First}(\alpha) = \{a \in V_t \mid \alpha \Rightarrow^* a\beta\} \cup \{\lambda \mid \text{if } \alpha \Rightarrow^* \lambda\}$
- $\text{Follow}(A) = \{a \in V_t \mid S \Rightarrow^+ \ldots Aa \ldots\} \cup \{\$ \mid \text{if } S \Rightarrow^+ \ldots A\ \$\}$

S:	start symbol
a:	a terminal symbol
A:	a non-terminal symbol
α,β:	a string composed of terminals and non-terminals (typically, α is the RHS of a production

$\Rightarrow$:	derived in 1 step
$\Rightarrow^*$:	derived in 0 or more steps
$\Rightarrow^+$:	derived in 1 or more steps

Slide courtesy: Milind Kulkarni

Computing first sets

- Terminal: First(a) = {a}
- Non-terminal: First(A)
  - Look at all productions for A

    $A \rightarrow X_1 X_2 \ldots X_k$
  - First(A) $\supseteq$ (First(X_1) - λ)
  - If $\lambda \in$ First(X_1), First(A) $\supseteq$ (First(X_2) - λ)
  - If λ is in First(X_i) for all i, then $\lambda \in$ First(A)
- Computing First(α): similar procedure to computing First(A)

A simple example

$S \rightarrow A\ B\ c\ \$$

$A \rightarrow x\ a\ A$

$A \rightarrow y\ a\ A$

$A \rightarrow c$

$B \rightarrow b$

$B \rightarrow \lambda$

- A sentence in the grammar:

  x a c c $

A simple example

S → A B c $

special "end of input" symbol

A → x a A

A → y a A

A → c

B → b

B → λ

- A sentence in the grammar:

  x a c c $

A simple example

$S \rightarrow A\ B\ c\ \$$

$A \rightarrow x\ a\ A$

$A \rightarrow y\ a\ A$

$A \rightarrow c$

$B \rightarrow b$

$B \rightarrow \lambda$

- A sentence in the grammar:

  x a c c $

Current derivation: S

A simple example

$S \rightarrow A\ B\ c\ \$$

$A \rightarrow x\ a\ A$

$A \rightarrow y\ a\ A$

$A \rightarrow c$

$B \rightarrow b$

$B \rightarrow \lambda$

- A sentence in the grammar:

  x a c c $

Current derivation: A B c $

Predict rule

A simple example

$S \rightarrow A\ B\ c\ \$$

Choose based on *first set* of rules

$A \rightarrow x\ a\ A$

$A \rightarrow y\ a\ A$

$A \rightarrow c$

$B \rightarrow b$

$B \rightarrow \lambda$

- A sentence in the grammar:

  x a c c $

Current derivation: x a A B c $

Predict rule *based on next token*

Slide courtesy: Milind Kulkarni

A simple example

$S \rightarrow A\ B\ c\ \$$

$A \rightarrow x\ a\ A$

$A \rightarrow y\ a\ A$

$A \rightarrow c$

$B \rightarrow b$

$B \rightarrow \lambda$

- A sentence in the grammar:

  x a c c $

Current derivation: x a A B c $

Match token

A simple example

$S \rightarrow A\ B\ c\ \$$

$A \rightarrow x\ a\ A$

$A \rightarrow y\ a\ A$

$A \rightarrow c$

$B \rightarrow b$

$B \rightarrow \lambda$

- A sentence in the grammar:

  x a c c $

Current derivation: x a A B c $

Match token

A simple example

$S \rightarrow A\ B\ c\ \$$

Choose based on *first set* of rules

$A \rightarrow x\ a\ A$
$A \rightarrow y\ a\ A$
$A \rightarrow c$

$B \rightarrow b$

$B \rightarrow \lambda$

- A sentence in the grammar:

  x a c c $

Current derivation: x a c B c $

Predict rule *based on next token*

A simple example

S → A B c $

A → x a A

A → y a A

A → c

B → b

B → λ

- A sentence in the grammar:

  x a c c $

Current derivation: x a c B c $

Match token

A simple example

$\boxed{S \rightarrow A\ B\ c\ \$}$

$A \rightarrow x\ a\ A$

Choose based on *follow set*

$A \rightarrow y\ a\ A$

$A \rightarrow c$

$\boxed{\begin{array}{l} B \rightarrow b \\ B \rightarrow \lambda \end{array}}$

- A sentence in the grammar:

  x a c c $

Current derivation: x a c λ c $

$\boxed{\text{Predict rule } \textit{based on next token}}$

A simple example

$S \rightarrow A\ B\ c\ \$$

$A \rightarrow x\ a\ A$

$A \rightarrow y\ a\ A$

$A \rightarrow c$

$B \rightarrow b$

$B \rightarrow \lambda$

- A sentence in the grammar:

  x a c c $

Current derivation: x a c c $

Match token

A simple example

$S \rightarrow A\ B\ c\ \$$

$A \rightarrow x\ a\ A$

$A \rightarrow y\ a\ A$

$A \rightarrow c$

$B \rightarrow b$

$B \rightarrow \lambda$

- A sentence in the grammar:

  x a c c $

Current derivation: x a c c $

Match token

Towards parser generators

- Key problem: as we read the source program, we need to decide what productions to use
- Step 1: find the tokens that can tell which production P (of the form $A \rightarrow X_1 X_2 \ldots X_m$) applies

$$\mathrm{Predict}(P) =$$

$$\begin{cases} \mathrm{First}(X_1 \ldots X_m) & \text{if } \lambda \notin \mathrm{First}(X_1 \ldots X_m) \\ (\mathrm{First}(X_1 \ldots X_m) - \lambda) \cup \mathrm{Follow}(A) & \text{otherwise} \end{cases}$$

- If next token is in Predict(P), then we should choose this production

First Set - Example

```
1) S -> ABc$
2) A -> xaA
3) A -> yaA
4) A -> c
5) B -> b
6) B -> λ
```

```
first (S) = { ? }
```

Think of all possible strings derivable from `S`. Get the first terminal symbol in those strings or λ if `S` derives λ

First Set - Example

```
1) S -> ABc$
2) A -> xaA
3) A -> yaA
4) A -> c
5) B -> b
6) B -> λ

first (S) = { x, y, c }
```

First Set - Example

```
1) S -> ABc$
2) A -> xaA
3) A -> yaA
4) A -> c
5) B -> b
6) B -> λ
```

```
first (S) = { x, y, c }
first (A) = {  ?  }
```

First Set - Example

```
1) S -> ABc$
2) A -> xaA
3) A -> yaA
4) A -> c
5) B -> b
6) B -> λ

first (S) = { x, y, c }
first (A) = { x, y, c }
```

First Set - Example

```
1) S -> ABc$
2) A -> xaA
3) A -> yaA
4) A -> c
5) B -> b
6) B -> λ
```

```
first (S) = { x, y, c }
first (A) = { x, y, c }
first (B) = {  ?  }
```

First Set - Example

```
1) S -> ABc$
2) A -> xaA
3) A -> yaA
4) A -> c
5) B -> b
6) B -> λ
```

```
first (S) = { x, y, c }
first (A) = { x, y, c }
first (B) = { b, λ }
```

Follow Set - Example

```
1) S -> ABc$
2) A -> xaA
3) A -> yaA
4) A -> c
5) B -> b
6) B -> λ
```

`follow (S) = { ? }`

Think of all strings **possible in the language** having the form `..Sa..` Get the following terminal symbol a after S in those strings or `$` if you get a string `..S$`

Follow Set - Example

```
1) S -> ABc$
2) A -> xaA
3) A -> yaA
4) A -> c
5) B -> b
6) B -> λ

follow (S) = {  }
```

Follow Set - Example

```
1) S -> ABc$
2) A -> xaA
3) A -> yaA
4) A -> c
5) B -> b
6) B -> λ

follow (S) = {   }
follow (A) = { ? }
```

Follow Set - Example

```
1) S -> ABc$
2) A -> xaA
3) A -> yaA
4) A -> c
5) B -> b
6) B -> λ

follow (S) = {  }
follow (A) = { b, c }      e.g. xaAbc$, xaAc$
```

Follow Set - Example

```
1) S -> ABc$
2) A -> xaA
3) A -> yaA
4) A -> c
5) B -> b
6) B -> λ
```

```
follow (S) = {  }
follow (A) = { b, c }        e.g. xaAbc$, xaAc$
```

What happens when you consider: `A -> xaA or A -> yaA ?`

Follow Set - Example

```
1) S -> ABc$
2) A -> xaA
3) A -> yaA
4) A -> c
5) B -> b
6) B -> λ
```

```
follow (S) = {  }
follow (A) = { b, c }        e.g. xaAbc$, xaAc$
```

What happens when you consider: `A -> xaA or A -> yaA ?`

- You will get string of the form `A=>`$^+$ `(xa)+A`
- But we need strings of the form: `..Aa..` or `..Ab.` or `..Ac..`

Follow Set - Example

```
1) S -> ABc$
2) A -> xaA
3) A -> yaA
4) A -> c
5) B -> b
6) B -> λ

follow (S) = {   }
follow (A) = { b, c }
follow (B) = { ? }
```

Follow Set - Example

```
1) S -> ABc$
2) A -> xaA
3) A -> yaA
4) A -> c
5) B -> b
6) B -> λ

follow (S) = {   }
follow (A) = { b, c }
follow (B) = { c }
```

Predict Set - Example

1) S -> ABc$
2) A -> xaA
3) A -> yaA
4) A -> c
5) B -> b
6) B -> λ

$$\mathrm{Predict}(P) =$$

$$\begin{cases} \mathrm{First}(X_1 \ldots X_m) & \text{if } \lambda \notin \mathrm{First}(X_1 \ldots X_m) \\ (\mathrm{First}(X_1 \ldots X_m) - \lambda) \cup \mathrm{Follow}(A) & \text{otherwise} \end{cases}$$

Predict (1) = { ? } = First(ABc$) if λ ∉ First(ABc$)

Predict Set - Example

```
1) S -> ABc$
2) A -> xaA
3) A -> yaA
4) A -> c
5) B -> b
6) B -> λ
```

	x	y	a	b	c	$
S	1	1			1	
A						
B						

Predict (1) = { x, y, c }

Predict Set - Example

1) S -> ABc$
2) A -> xaA
3) A -> yaA
4) A -> c
5) B -> b
6) B -> λ

	x	y	a	b	c	$
S	1	1			1	
A						
B						

Predict (1) = { x, y, c }

Predict (2) = { ? } = First(xaA) if $\lambda \notin$ First(xaA)

Predict Set - Example

```
1) S -> ABc$
2) A -> xaA
3) A -> yaA
4) A -> c
5) B -> b
6) B -> λ
```

	x	y	a	b	c	$
S	1	1			1	
A	2					
B						

```
Predict (1) = { x, y, c }
Predict (2) = { x }
```

Predict Set - Example

```
1) S -> ABc$
2) A -> xaA
3) A -> yaA
4) A -> c
5) B -> b
6) B -> λ
```

	x	y	a	b	c	$
S	1	1			1	
A	2					
B						

```
Predict (1) = { x, y, c }
Predict (2) = { x }
Predict (3) = { ? }
```

= First(yaA) if $\lambda \notin$ First(yaA)

Predict Set - Example

```
1) S -> ABc$
2) A -> xaA
3) A -> yaA
4) A -> c
5) B -> b
6) B -> λ
```

	x	y	a	b	c	$
S	1	1			1	
A	2	3				
B						

```
Predict (1) = { x, y, c }
Predict (2) = { x }
Predict (3) = { y }
```

Predict Set - Example

1) S -> ABc$
2) A -> xaA
3) A -> yaA
4) A -> c
5) B -> b
6) B -> λ

	x	y	a	b	c	$
S	1	1			1	
A	2	3				
B						

Predict (1) = { x, y, c }
Predict (2) = { x }
Predict (3) = { y }
Predict (4) = { ? } = First(c) if $\lambda \notin$ First(c)

Predict Set - Example

```
1) S -> ABc$
2) A -> xaA
3) A -> yaA
4) A -> c
5) B -> b
6) B -> λ
```

	x	y	a	b	c	$
S	1	1			1	
A	2	3			4	
B						

```
Predict (1) = { x, y, c }
Predict (2) = { x }
Predict (3) = { y }
Predict (4) = { c }
```

Predict Set - Example

1) S -> ABc$
2) A -> xaA
3) A -> yaA
4) A -> c
5) B -> b
6) B -> λ

	x	y	a	b	c	$
S	1	1			1	
A	2	3			4	
B						

Predict (1) = { x, y, c }
Predict (2) = { x }
Predict (3) = { y }
Predict (4) = { c }
Predict (5) = { ? }   = First(b) if λ ∉ First(b)

Predict Set - Example

```
1) S -> ABc$
2) A -> xaA
3) A -> yaA
4) A -> c
5) B -> b
6) B -> λ
```

	x	y	a	b	c	$
S	1	1			1	
A	2	3			4	
B				5		

```
Predict (1) = { x, y, c }
Predict (2) = { x }
Predict (3) = { y }
Predict (4) = { c }
Predict (5) = { b }
```

Predict Set - Example

```
1) S -> ABc$
2) A -> xaA
3) A -> yaA
4) A -> c
5) B -> b
6) B -> λ
```

	x	y	a	b	c	$
S	1	1			1	
A	2	3			4	
B				5		

```
Predict (1) = { x, y, c }
Predict (2) = { x }
Predict (3) = { y }
Predict (4) = { c }
Predict (5) = { b }
Predict (6) = { ? }
```

$\text{Predict}(P) =$

$$\begin{cases} \text{First}(X_1 \ldots X_m) & \text{if } \lambda \notin \text{First}(X_1 \ldots X_m) \\ (\text{First}(X_1 \ldots X_m) - \lambda) \cup \text{Follow}(A) & \text{otherwise} \end{cases}$$

= First(λ) ?

Predict Set - Example

```
1) S -> ABc$
2) A -> xaA
3) A -> yaA
4) A -> c
5) B -> b
6) B -> λ
```

	x	y	a	b	c	$
S	1	1			1	
A	2	3			4	
B				5		

```
Predict (1) = { x, y, c }
Predict (2) = { x }
Predict (3) = { y }
Predict (4) = { c }
Predict (5) = { b }
Predict (6) = { ? }
```

$$\text{Predict}(P) = \begin{cases} \text{First}(X_1 \ldots X_m) & \text{if } \lambda \notin \text{First}(X_1 \ldots X_m) \\ \boxed{(\text{First}(X_1 \ldots X_m) - \lambda) \cup \text{Follow}(A) \quad \text{otherwise}} \end{cases}$$

= $\cancel{\text{First}(\lambda)}$? Follow(B)

Predict Set - Example

```
1) S -> ABc$
2) A -> xaA
3) A -> yaA
4) A -> c
5) B -> b
6) B -> λ
```

	x	y	a	b	c	$
S	1	1			1	
A	2	3			4	
B				5	6	

```
Predict (1) = { x, y, c }
Predict (2) = { x }
Predict (3) = { y }
Predict (4) = { c }
Predict (5) = { b }
Predict (6) = { c }
```

Computing Parse-Table

```
1) S -> ABc$
2) A -> xaA
3) A -> yaA
4) A -> c
5) B -> b
6) B -> λ
```

	x	y	a	b	c	$
S	1	1			1	
A	2	3			4	
B				5	6	

```
first (S) = {x, y, c}
first (A) = {x, y, c}
first(B) = {b, λ}
```

```
follow (S) = {}
follow (A) = {b, c}
follow(B) = {c}
```

```
P(1) = {x,y,c}
P(2) = {x}
P(3) = {y}
P(4) = {c}
P(5) = {b}
P(6) = {c}
```

Parsing using stack-based model

- How do we use the Parse Table constructed?

Top-Down Parsing - Example

`string: xacc$`

Stack	Rem. Input	Action
?	xacc$	?

What do you put on the stack?

Top-Down Parsing - Example

`string: xacc$`

Stack	Rem. Input	Action
?	xacc$	?

What do you put on the stack? – strings that you derive

Top-Down Parsing - Example

string: xacc$

Stack*	**Rem. Input**	**Action**
S	xacc$	?

Top-down parsing. So, start with S.

* Stack top is on the left-side (first symbol) of the column

Top-Down Parsing - Example

`string: xacc$`

Stack*	**Rem. Input**	**Action**
S	xacc$	?

Top-down parsing. So, start with S.

What action do you take when stack-top has symbol S and the string to be matched has terminal x in front?

* Stack top is on the left-side (first symbol) of the column

Top-Down Parsing - Example

string: xacc$

Stack*	**Rem. Input**	**Action**
S	xacc$	Predict(1) S->ABc$

Top-down parsing. So, start with S.

What action do you take when stack-top has symbol S and the string to be matched has terminal x in front? – consult parse table

	x	y	a	b	c	$
S	1	1			1	
A	2	3			4	
B				5	6	

* Stack top is on the left-side (first symbol) of the column

Top-Down Parsing - Example

string: xacc$

Stack*	Rem. Input	Action
S	xacc$	Predict(1) S->ABc$
ABc$		

	x	y	a	b	c	$
S	1	1			1	
A	2	3			4	
B				5	6	

* Stack top is on the left-side (first symbol) of the column

Top-Down Parsing - Example

string: xacc$

Stack*	Rem. Input	Action
S	xacc$	Predict(1) S->ABc$
ABc$	xacc$	

What action do you take when stack-top has symbol A and the string to be matched has terminal x in front? – consult parse table

	x	y	a	b	c	$
S	1	1			1	
A	2	3			4	
B				5	6	

* Stack top is on the left-side (first symbol) of the column

Top-Down Parsing - Example

string: xacc$

Stack*	Rem. Input	Action
S	xacc$	Predict(1) S->ABc$
ABc$	xacc$	Predict(2) A->xaA

What action do you take when stack-top has symbol A and the string to be matched has terminal x in front? – consult parse table

	x	y	a	b	c	$
S	1	1			1	
A	2	3			4	
B				5	6	

* Stack top is on the left-side (first symbol) of the column

Top-Down Parsing - Example

string: xacc$

Stack*	Rem. Input	Action
S	xacc$	Predict(1) S->ABc$
ABc$	xacc$	Predict(2) A->xaA
xaABc$		

	x	y	a	b	c	$
S	1	1			1	
A	2	3			4	
B				5	6	

* Stack top is on the left-side (first symbol) of the column

Top-Down Parsing - Example

string: xacc$

Stack*	**Rem. Input**	**Action**
S	xacc$	Predict(1) S->ABc$
ABc$	xacc$	Predict(2) A->xaA
xaABc$	xacc$	?

	x	y	a	b	c	$
S	1	1			1	
A	2	3			4	
B				5	6	

* Stack top is on the left-side (first symbol) of the column

Top-Down Parsing - Example

string: xacc$

Stack*	Rem. Input	Action
S	xacc$	Predict(1) S->ABc$
ABc$	xacc$	Predict(2) A->xaA
xaABc$	xacc$	match(x)

	x	y	a	b	c	$
S	1	1			1	
A	2	3			4	
B				5	6	

* Stack top is on the left-side (first symbol) of the column

Top-Down Parsing - Example

string: xacc$

Stack*	Rem. Input	Action
S	xacc$	Predict(1) S->ABc$
ABc$	xacc$	Predict(2) A->xaA
xaABc$	xacc$	match(x)
aABc$	acc$	match(a)

	x	y	a	b	c	$
S	1	1			1	
A	2	3			4	
B				5	6	

* Stack top is on the left-side (first symbol) of the column

Top-Down Parsing - Example

string: xacc$

Stack*	**Rem. Input**	**Action**
S	xacc$	Predict(1) S->ABc$
ABc$	xacc$	Predict(2) A->xaA
xaABc$	xacc$	match(x)
aABc$	acc$	match(a)
ABc$	cc$	?

	x	y	a	b	c	$
S	1	1			1	
A	2	3			4	
B				5	6	

* Stack top is on the left-side (first symbol) of the column

Top-Down Parsing - Example

	x	y	a	b	c	$
S	1	1			1	
A	2	3			4	
B				5	6	

string: xacc$

Stack*	**Rem. Input**	**Action**
S	xacc$	Predict(1) S->ABc$
ABc$	xacc$	Predict(2) A->xaA
xaABc$	xacc$	match(x)
aABc$	acc$	match(a)
ABc$	cc$	Predict(4) A->c

* Stack top is on the left-side (first symbol) of the column

Top-Down Parsing - Example

	x	y	a	b	c	$
S	1	1			1	
A	2	3			4	
B				5	6	

string: xacc$

Stack*	Rem. Input	Action
S	xacc$	Predict(1) S->ABc$
ABc$	xacc$	Predict(2) A->xaA
xaABc$	xacc$	match(x)
aABc$	acc$	match(a)
ABc$	cc$	Predict(4) A->c
cBc$		

* Stack top is on the left-side (first symbol) of the column

Top-Down Parsing - Example

	x	y	a	b	c	$
S	1	1			1	
A	2	3			4	
B				5	6	

string: xacc$

Stack*	**Rem. Input**	**Action**
S	xacc$	Predict(1) S->ABc$
ABc$	xacc$	Predict(2) A->xaA
xaABc$	xacc$	match(x)
aABc$	acc$	match(a)
ABc$	cc$	Predict(4) A->c
cBc$	cc$	?

* Stack top is on the left-side (first symbol) of the column

Top-Down Parsing - Example

	x	y	a	b	c	$
S	1	1			1	
A	2	3			4	
B				5	6	

string: xacc$

Stack*	Rem. Input	Action
S	xacc$	Predict(1) S->ABc$
ABc$	xacc$	Predict(2) A->xaA
xaABc$	xacc$	match(x)
aABc$	acc$	match(a)
ABc$	cc$	Predict(4) A->c
cBc$	cc$	match(c)

* Stack top is on the left-side (first symbol) of the column

Top-Down Parsing - Example

	x	y	a	b	c	$
S	1	1			1	
A	2	3			4	
B				5	6	

string: xacc$

Stack*	**Rem. Input**	**Action**
S	xacc$	Predict(1) S->ABc$
ABc$	xacc$	Predict(2) A->xaA
xaABc$	xacc$	match(x)
aABc$	acc$	match(a)
ABc$	cc$	Predict(4) A->c
cBc$	cc$	match(c)
Bc$	c$	?

* Stack top is on the left-side (first symbol) of the column

Top-Down Parsing - Example

	x	y	a	b	c	$
S	1	1			1	
A	2	3			4	
B				5	6	

string: xacc$

Stack*	**Rem. Input**	**Action**
S	xacc$	Predict(1) S->ABc$
ABc$	xacc$	Predict(2) A->xaA
xaABc$	xacc$	match(x)
aABc$	acc$	match(a)
ABc$	cc$	Predict(4) A->c
cBc$	cc$	match(c)
Bc$	c$	Predict(6) B->λ

* Stack top is on the left-side (first symbol) of the column

Top-Down Parsing - Example

	x	y	a	b	c	$
S	1	1			1	
A	2	3			4	
B				5	6	

string: xacc$

Stack*	**Rem. Input**	**Action**
S	xacc$	Predict(1) S->ABc$
ABc$	xacc$	Predict(2) A->xaA
xaABc$	xacc$	match(x)
aABc$	acc$	match(a)
ABc$	cc$	Predict(4) A->c
cBc$	cc$	match(c)
Bc$	c$	Predict(6) B->λ
c$		

* Stack top is on the left-side (first symbol) of the column

Top-Down Parsing - Example

	x	y	a	b	c	$
S	1	1			1	
A	2	3			4	
B				5	6	

string: xacc$

Stack*	Rem. Input	Action
S	xacc$	Predict(1) S->ABc$
ABc$	xacc$	Predict(2) A->xaA
xaABc$	xacc$	match(x)
aABc$	acc$	match(a)
ABc$	cc$	Predict(4) A->c
cBc$	cc$	match(c)
Bc$	c$	Predict(6) B->λ
c$	c$	?

* Stack top is on the left-side (first symbol) of the column

Top-Down Parsing - Example

	x	y	a	b	c	$
S	1	1			1	
A	2	3			4	
B				5	6	

string: xacc$

Stack*	Rem. Input	Action
S	xacc$	Predict(1) S->ABc$
ABc$	xacc$	Predict(2) A->xaA
xaABc$	xacc$	match(x)
aABc$	acc$	match(a)
ABc$	cc$	Predict(4) A->c
cBc$	cc$	match(c)
Bc$	c$	Predict(6) B->λ
c$	c$	match(c)

* Stack top is on the left-side (first symbol) of the column

Top-Down Parsing - Example

	x	y	a	b	c	$
S	1	1			1	
A	2	3			4	
B				5	6	

string: xacc$

Stack*	Rem. Input	Action
S	xacc$	Predict(1) S->ABc$
ABc$	xacc$	Predict(2) A->xaA
xaABc$	xacc$	match(x)
aABc$	acc$	match(a)
ABc$	cc$	Predict(4) A->c
cBc$	cc$	match(c)
Bc$	c$	Predict(6) B->λ
c$	c$	match(c)
$	$	Done!

* Stack top is on the left-side (first symbol) of the column

Identifying LL(1) Grammar

- What we saw was an example of LL(1) Grammar
  - Scan input **L**eft-to-right, produce **L**eft-most derivation with 1 symbol look-ahead

Identifying LL(1) Grammar

- What we saw was an example of LL(1) Grammar
  - Scan input **L**eft-to-right, produce **L**eft-most derivation with 1 symbol look-ahead
- Not all Grammars are LL(1)

A Grammar is LL(1) iff for a production A -> α | β, where

α and β are distinct:

1. For no terminal `a` do both α and β derive strings beginning with `a` (i.e. no common prefix)
2. At most one of α and β can derive an empty string
3. If $\beta \overset{*}{\Rightarrow} \epsilon,$ then α does not derive any string beginning with a terminal in `Follow(A)`. If $\alpha \overset{*}{\Rightarrow} \epsilon,$ then β does not derive any string beginning with a terminal in `Follow(A)`

Example (Left Factoring)

- Consider

  <stmt> → if <expr> then <stmt list> endif

  <stmt> → if <expr> then <stmt list> else <stmt list> endif

- This is not LL(1) (why?)
- We can turn this in to

  <stmt> → if <expr> then <stmt list> <if suffix>

  <if suffix> → endif

  <if suffix> → else <stmt list> endif

Example (Left Factoring)

- Consider

  <stmt> → if <expr> then <stmt list> endif

  <stmt> → if <expr> then <stmt list> else <stmt list> endif

- This is not LL(1) (why?)
- We can turn this in to

  <stmt> → if <expr> then <stmt list> <if suffix>

  <if suffix> → endif

  <if suffix> → else <stmt list> endif

Left Factoring

A -> α β | α μ

$$\Downarrow$$

A -> α N

N -> β

N -> μ

Left recursion

- *Left recursion* is a problem for LL(1) parsers
  - LHS is also the first symbol of the RHS
- Consider:

  $E \rightarrow E + T$

- What would happen with the stack-based algorithm?

Left recursion

- *Left recursion* is a problem for LL(1) parsers
  - LHS is also the first symbol of the RHS
- Consider:

  $E \rightarrow E + T$

- What would happen with the stack-based algorithm?

  E
  E + T
  E + T + T
  E + T + T + T

Eliminating Left Recursion

A -> A α | β

$$\Downarrow$$

A -> NT
N -> β
T -> αT
T -> λ

Eliminating Left Recursion

E -> E + T

⇩

E -> E1 Etail
E1 -> T
Etail -> + T Etail
Etail -> λ

LL(k) parsers

- Can look ahead more than one symbol at a time
  - *k*-symbol lookahead requires extending first and follow sets
  - 2-symbol lookahead can distinguish between more rules:

    $A \rightarrow ax \mid ay$
- More lookahead leads to more powerful parsers
- What are the downsides?

Are all grammars LL(k)?

- No! Consider the following grammar:

$$
\begin{aligned}
S &\rightarrow E \\
E &\rightarrow (E + E) \\
E &\rightarrow (E - E) \\
E &\rightarrow x
\end{aligned}
$$

- When parsing E, how do we know whether to use rule 2 or 3?
  - Potentially unbounded number of characters before the distinguishing '+' or '–' is found
  - No amount of lookahead will help!

LL(k)? - Example

1) S - > E
2) E -> (E+E)
3) E -> (E-E)
4) E -> x

string: ((x+x))$

Stack*	Rem. Input	Action
S	((x+x))$	Predict(1) S->E
E		Predict(2) or Predict(3)?

LL(1)

	(	+	-	)	x
S	1				1
E	2,3				4

LL(2)

	((	+(	-(	)$	(x
S	1				1
E	2,3				4

In real languages?

- Consider the if-then-else problem
- `if x then y else z`
- Problem: else is optional
- `if a then if b then c else d`
    - Which `if` does the `else` belong to?
- This is analogous to a "bracket language": $[^i\,]^j$ $(i \geq j)$

$$
\begin{aligned}
S &\rightarrow [\,S\,C \\
S &\rightarrow \lambda \\
C &\rightarrow] \\
C &\rightarrow \lambda
\end{aligned}
$$

[[ ] can be parsed: SSλC or SSCλ
(it's ambiguous!)

Solving the if-then-else problem

- The ambiguity exists at the language level. To fix, we need to define the semantics properly
  - "] matches nearest unmatched ["
  - This is the rule C uses for if-then-else
  - What if we try this?

$$
\begin{aligned}
S &\rightarrow [\ S \\
S &\rightarrow S1 \\
S1 &\rightarrow [\ S1\ ] \\
S1 &\rightarrow \lambda
\end{aligned}
$$

This grammar is still not LL(1) (or LL(k) for any k!)

Two possible fixes

- If there is an ambiguity, prioritize one production over another
  - e.g., if C is on the stack, always match "]" before matching "λ"

$$
\begin{aligned}
S &\rightarrow [\ S\ C \\
S &\rightarrow \lambda \\
C &\rightarrow] \\
C &\rightarrow \lambda
\end{aligned}
$$

- Another option: change the language!
  - e.g., all if-statements need to be closed with an endif

$$
\begin{aligned}
S &\rightarrow \text{if } S\ E \\
S &\rightarrow \text{other} \\
E &\rightarrow \text{else } S \text{ endif} \\
E &\rightarrow \text{endif}
\end{aligned}
$$

Parsing if-then-else

- What if we don't want to change the language?
  - C does not require { } to delimit single-statement blocks
- To parse if-then-else, *we need to be able to look ahead at the entire rhs of a production* before deciding which production to use
  - In other words, we need to determine how many "]" to match before we start matching "["s
- *LR parsers* can do this!

Bottom-up Parsing

- More general than top-down parsing
- Used in most parser-generator tools
- Need not have left-factored grammars (i.e. can have left recursion)
- E.g. can work with the bracket language

Bottom-up Parsing

- Reduce a string to start symbol by reverse 'inverting' productions

Bottom-up Parsing

- Reduce a string to start symbol by reverse 'inverting' productions

```
id * id + id
```

```
E -> T + E
E -> T
T -> id * T
T -> id
```

Bottom-up Parsing

- Reduce a string to start symbol by reverse 'inverting' productions

```
id * id + id
id * T + id
```

```
E -> T + E
E -> T
T -> id * T
T -> id
```

Bottom-up Parsing

- Reduce a string to start symbol by reverse 'inverting' productions

```
id * id + id
id * T + id
T + id
```

```
E -> T + E
E -> T
T -> id * T
T -> id
```

Bottom-up Parsing

- Reduce a string to start symbol by reverse 'inverting' productions

```
id * id + id
id * T + id
T + id
T + T
```

```
E -> T + E
E -> T
T -> id * T
T -> id
```

Bottom-up Parsing

- Reduce a string to start symbol by reverse 'inverting' productions

```
E -> T + E
E -> T
T -> id * T
T -> id
```

```
id * id + id
id * T + id
T + id
T + T
T + E
```

Bottom-up Parsing

- Reduce a string to start symbol by reverse 'inverting' productions

```
E -> T + E
E -> T
T -> id * T
T -> id
```

```
id * id + id
id * T + id
T + id
T + T
T + E
E
```

Bottom-up Parsing

- Reduce a string to start symbol by reverse 'inverting' productions

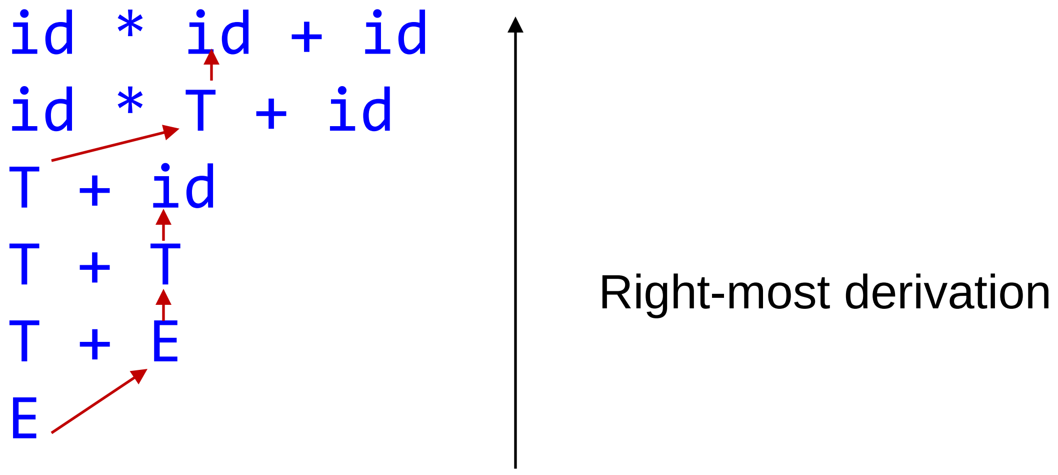
```
E -> T + E
E -> T
T -> id * T
T -> id
```

```
id * id + id
id * T + id
T + id
T + T
T + E
E
```

Bottom-up Parsing

- Reduce a string to start symbol by reverse 'inverting' productions

```
E -> T + E
E -> T
T -> id * T
T -> id
```



```
id * id + id
id * T + id
T + id
T + T
T + E
E
```

Right-most derivation

Bottom-up Parsing

- Scan the input left-to-right and shift tokens – put them on the stack.

```
| id * id + id              E -> T + E
                            E -> T
id | * id + id              T -> id * T
                            T -> id
id * | id + id

id * id | + id
```

Bottom-up Parsing

- Replace a set of symbols at the top of the stack that are RHS of a production. Put the LHS of the production on stack – Reduce

```
| id * id + id

id | * id + id

id * | id + id

id * id | + id
```

```
E -> T + E
E -> T
T -> id * T
T -> id
```

Bottom-up Parsing

- Did not discuss when and why a particular production was chosen

```
id * id + id
id * T + id
```

```
E -> T + E
E -> T
T -> id * T
T -> id
```

- *i.e. why replace the id highlighted in input string?*

LR Parsers

- Parser which does a Left-to-right, Right-most derivation
    - Rather than parse top-down, like LL parsers do, parse bottom-up, starting from leaves
- Basic idea: put tokens on a stack until an entire production is found
- Issues:
    - Recognizing the endpoint of a production
    - Finding the length of a production (RHS)
    - Finding the corresponding nonterminal (the LHS of the production)

Data structures

- At each state, given the next token,
  - A *goto table* defines the successor state
  - An *action table* defines whether to
    - *shift* – put the next state and token on the stack
    - *reduce* – an RHS is found; process the production
    - *terminate* – parsing is complete

Simple example

1. P → S
2. S → x ; S
3. S → e

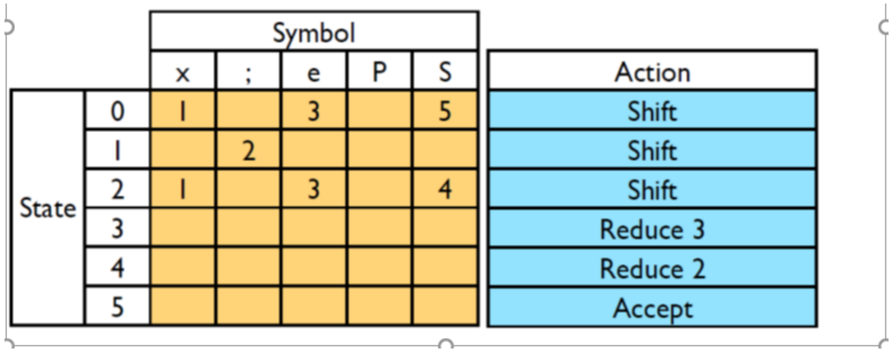
		Symbol					
		x	;	e	P	S	Action
State	0	1		3		5	Shift
	1		2				Shift
	2	1		3		4	Shift
	3						Reduce 3
	4						Reduce 2
	5						Accept

Example

I) P->S
II) S->x;S
III) S->e

<u>Input string</u>

x;x;e



		Symbol					
		x	;	e	P	S	Action
State	0	1		3		5	Shift
	1		2				Shift
	2	1		3		4	Shift
	3						Reduce 3
	4						Reduce 2
	5						Accept

Step	Parse Stack	Rem. Input	Parser Action
1	0	x;x;e	?

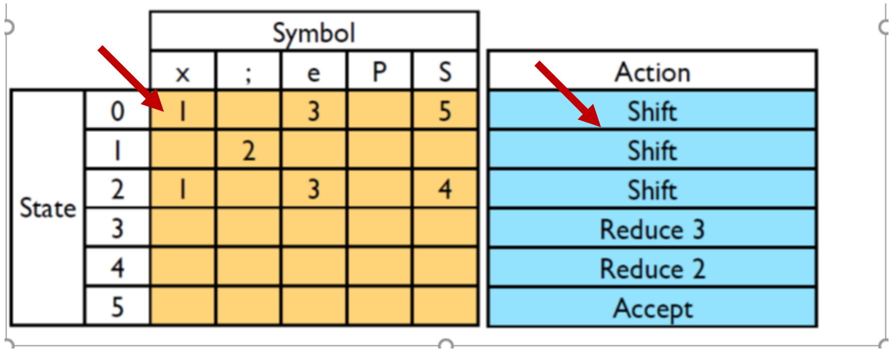
Start with state 0

Example

I) P->S
II) S->x;S
III) S->e

Input string

x;x;e



		Symbol					Action
		x	;	e	P	S	Action
State	0	1		3		5	Shift
	1		2				Shift
	2	1		3		4	Shift
	3						Reduce 3
	4						Reduce 2
	5						Accept

Step	Parse Stack	Rem. Input	Parser Action
1	0	x;x;e	Shift(1)

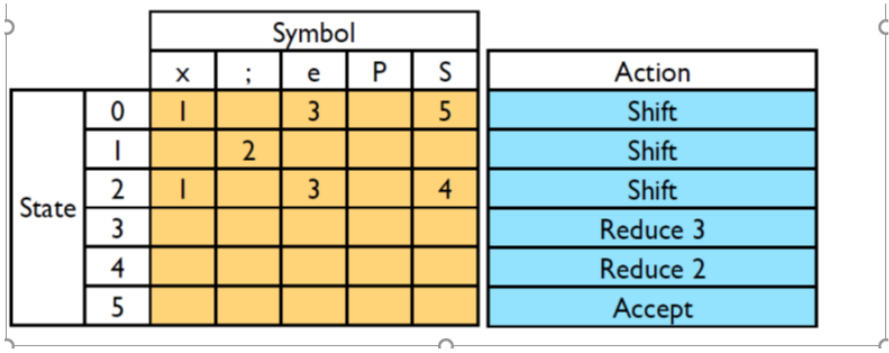
CS406, IIT Dharwad

Example

I) P->S
II) S->x;S
III) S->e

<u>Input string</u>

x;x;e



		Symbol					Action
		x	;	e	P	S	
State	0	1		3		5	Shift
	1		2				Shift
	2	1		3		4	Shift
	3						Reduce 3
	4						Reduce 2
	5						Accept

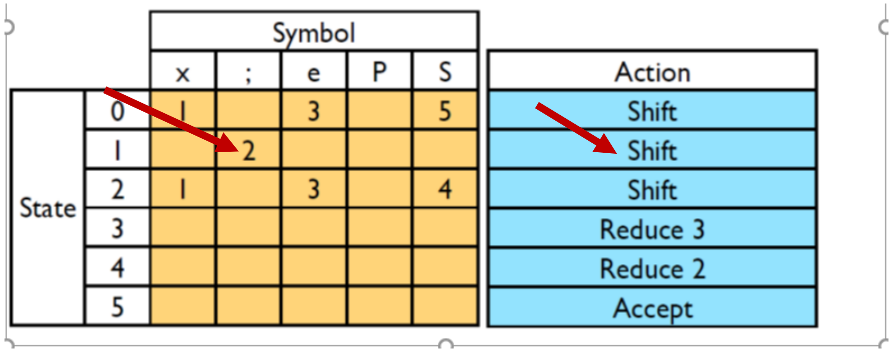
Step	Parse Stack	Rem. Input	Parser Action
1	0	x;x;e	Shift(1)
2	0 1	;x;e	?

Example

I) P->S
II) S->x;S
III) S->e

<u>Input string</u>

x;x;e



		Symbol					
		x	;	e	P	S	Action
State	0	1		3		5	Shift
	1		2				Shift
	2	1		3		4	Shift
	3						Reduce 3
	4						Reduce 2
	5						Accept

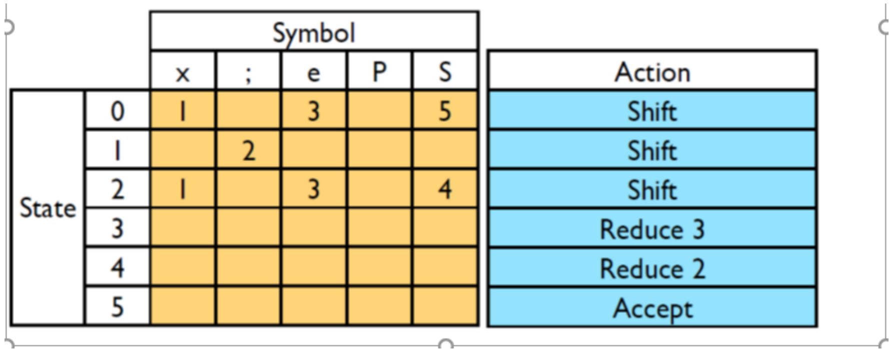
Step	Parse Stack	Rem. Input	Parser Action
1	0	x;x;e	Shift(1)
2	0 1	;x;e	Shift(2)

Example

I) P->S
II) S->x;S
III) S->e

<u>Input string</u>

x;x;e



		Symbol					
		x	;	e	P	S	Action
State	0	1		3		5	Shift
	1		2				Shift
	2	1		3		4	Shift
	3						Reduce 3
	4						Reduce 2
	5						Accept

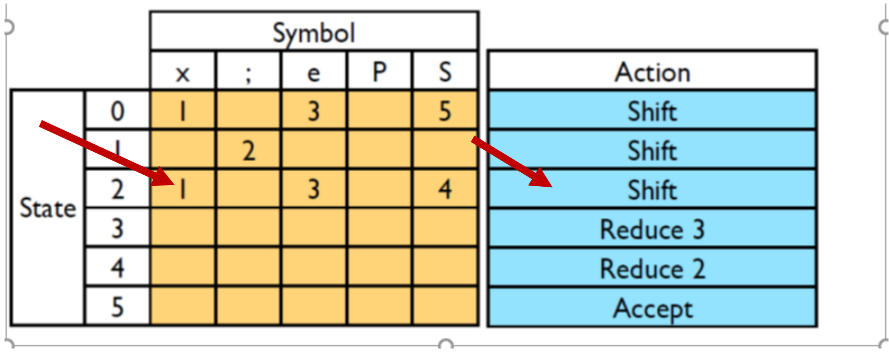
Step	**Parse Stack**	**Rem. Input**	**Parser Action**
1	0	x;x;e	Shift(1)
2	0 1	;x;e	Shift(2)
3	0 1 2	x;e	**?**

Example

I) P->S
II) S->x;S
III) S->e

Input string

x;x;e



		Symbol					Action
		x	;	e	P	S	
State	0	1		3		5	Shift
	1		2				Shift
	2	1		3		4	Shift
	3						Reduce 3
	4						Reduce 2
	5						Accept

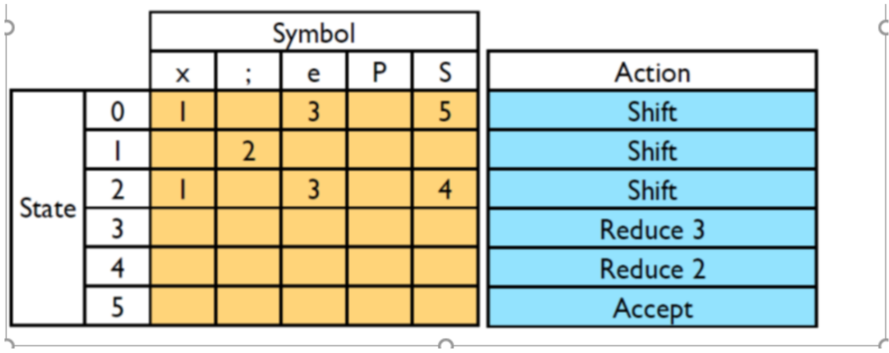
Step	Parse Stack	Rem. Input	Parser Action
1	0	x;x;e	Shift(1)
2	0 1	;x;e	Shift(2)
3	0 1 2	x;e	Shift(1)

Example

```
I) P->S
II) S->x;S
III) S->e
```

Input string

`x;x;e`



		Symbol					
		x	;	e	P	S	Action
State	0	1		3		5	Shift
	1		2				Shift
	2	1		3		4	Shift
	3						Reduce 3
	4						Reduce 2
	5						Accept

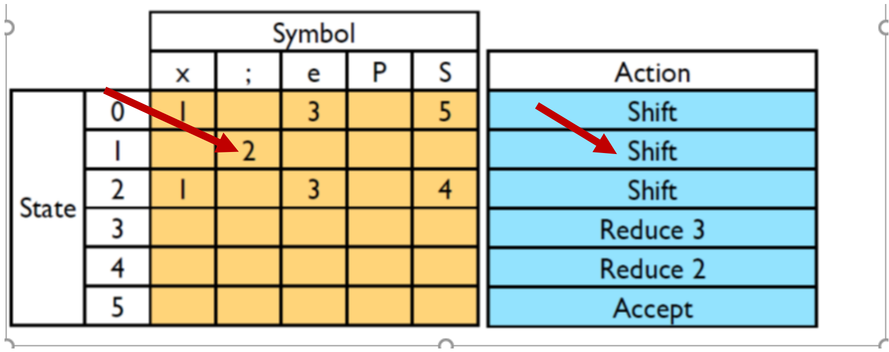
Step	Parse Stack	Rem. Input	Parser Action
1	0	x;x;e	Shift(1)
2	0 1	;x;e	Shift(2)
3	0 1 2	x;e	Shift(1)
4	0 1 2 1	;e	?

Example

I) P->S
II) S->x;S
III) S->e

<u>Input string</u>

x;x;e



		Symbol					
		x	;	e	P	S	Action
State	0	1		3		5	Shift
	1		2				Shift
	2	1		3		4	Shift
	3						Reduce 3
	4						Reduce 2
	5						Accept

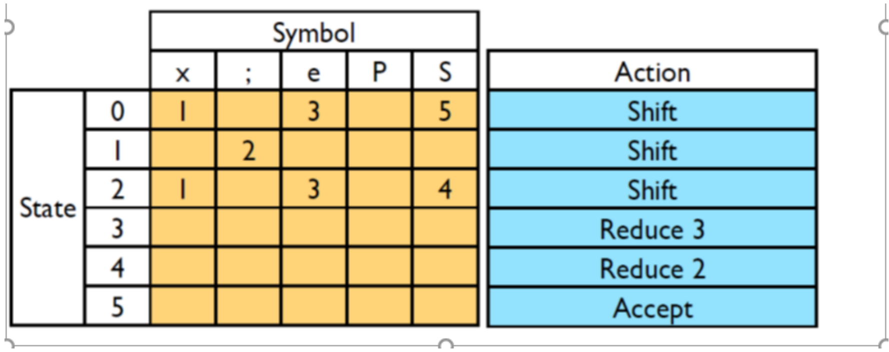
Step	Parse Stack	Rem. Input	Parser Action
1	0	x;x;e	Shift(1)
2	0 1	;x;e	Shift(2)
3	0 1 2	x;e	Shift(1)
4	0 1 2 1	;e	Shift(2)

Example

I) P->S
II) S->x;S
III) S->e

<u>Input string</u>

x;x;e



		Symbol					Action
		x	;	e	P	S	
State	0	1		3		5	Shift
	1		2				Shift
	2	1		3		4	Shift
	3						Reduce 3
	4						Reduce 2
	5						Accept

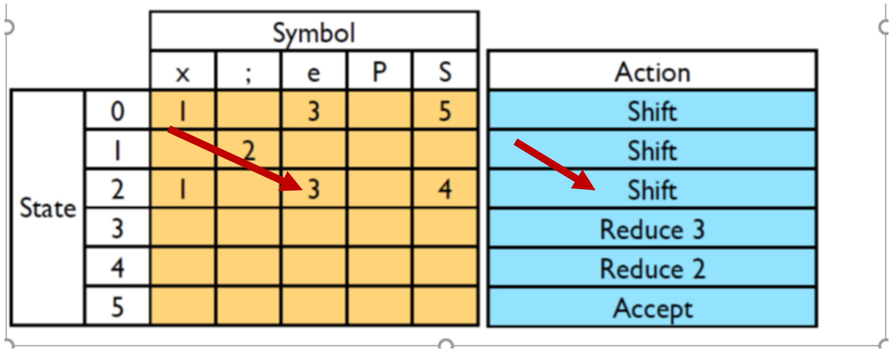
Step	**Parse Stack**	**Rem. Input**	**Parser Action**
1	0	x;x;e	Shift(1)
2	0 1	;x;e	Shift(2)
3	0 1 2	x;e	Shift(1)
4	0 1 2 1	;e	Shift(2)
5	0 1 2 1 2	e	**?**

Example

I) P->S
II) S->x;S
III) S->e

<u>Input string</u>

x;x;e



		Symbol					Action
		x	;	e	P	S	
State	0	1		3		5	Shift
	1		2				Shift
	2	1		3		4	Shift
	3						Reduce 3
	4						Reduce 2
	5						Accept

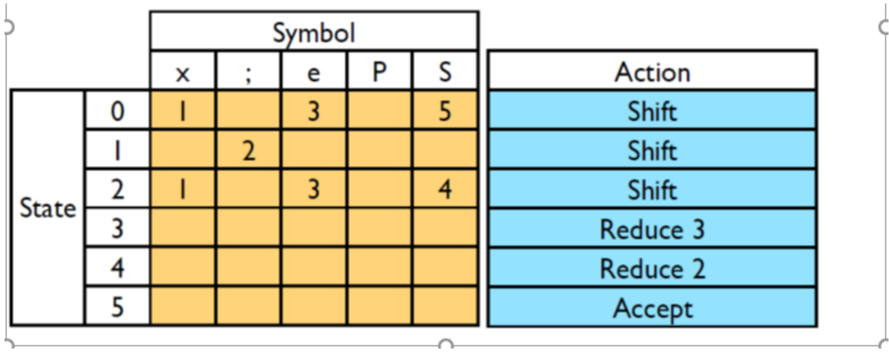
Step	**Parse Stack**	**Rem. Input**	**Parser Action**
1	0	x;x;e	Shift(1)
2	0 1	;x;e	Shift(2)
3	0 1 2	x;e	Shift(1)
4	0 1 2 1	;e	Shift(2)
5	0 1 2 1 2	e	Shift(3)

Example

```
I) P->S
II) S->x;S
III) S->e
```

Input string

x;x;e



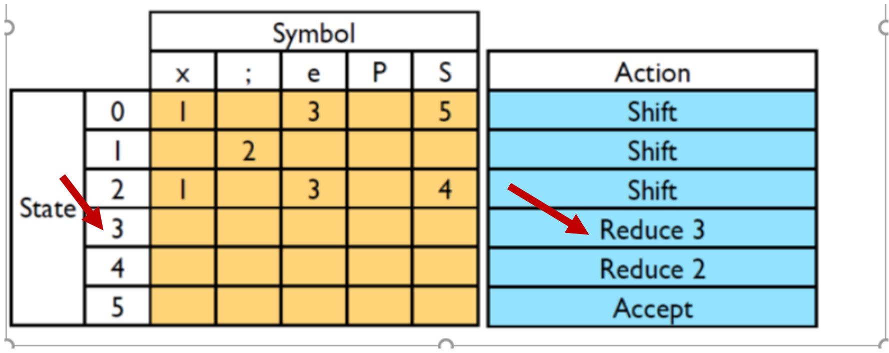
		Symbol					
		x	;	e	P	S	Action
State	0	1		3		5	Shift
	1		2				Shift
	2	1		3		4	Shift
	3						Reduce 3
	4						Reduce 2
	5						Accept

Step	Parse Stack	Rem. Input	Parser Action
1	0	x;x;e	Shift(1)
2	0 1	;x;e	Shift(2)
3	0 1 2	x;e	Shift(1)
4	0 1 2 1	;e	Shift(2)
5	0 1 2 1 2	e	Shift(3)
6	0 1 2 1 2 3		?

Example

I) P->S
II) S->x;S
III) S->e

Input string
x;x;e



		Symbol					Action
		x	;	e	P	S	
State	0	1		3		5	Shift
	1		2				Shift
	2	1		3		4	Shift
	3						Reduce 3
	4						Reduce 2
	5						Accept

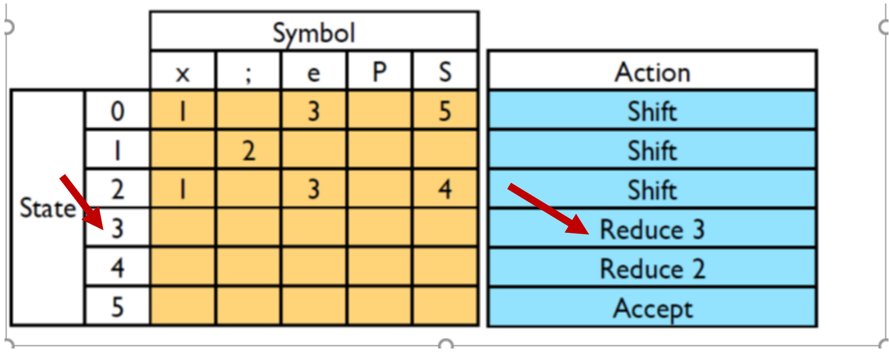
Step	Parse Stack	Rem. Input	Parser Action
1	0	x;x;e	Shift(1)
2	0 1	;x;e	Shift(2)
3	0 1 2	x;e	Shift(1)
4	0 1 2 1	;e	Shift(2)
5	0 1 2 1 2	e	Shift(3)
6	0 1 2 1 2 3		Reduce 3

Example

```
I) P->S
II) S->x;S
III) S->e
```

Input string

`x;x;e`



		Symbol					Action
		x	;	e	P	S	
State	0	1		3		5	Shift
	1		2				Shift
	2	1		3		4	Shift
	3						Reduce 3
	4						Reduce 2
	5						Accept

Step	Parse Stack	Rem. Input	Parser Action
1	0	x;x;e	Shift(1)
2	0 1	;x;e	Shift(2)
3	0 1 2	x;e	Shift(1)
4	0 1 2 1	;e	Shift(2)
5	0 1 2 1 2	e	Shift(3)
6	0 1 2 1 2 3		Reduce 3
7	0 1 2 1 2		

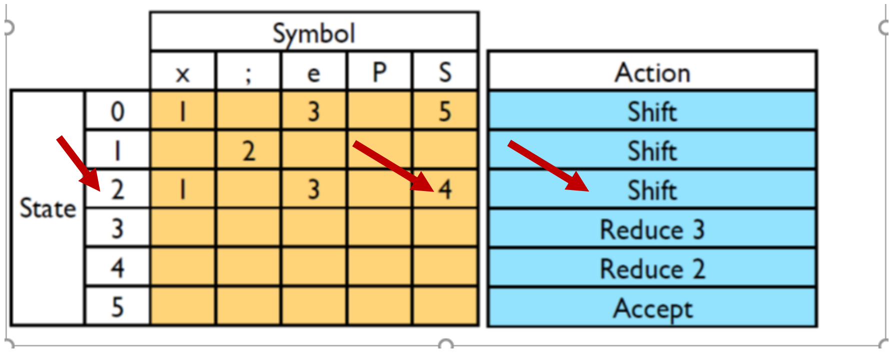
- *Look at rule III and pop 1 symbol of the stack because RHS of rule III has just 1 symbol*

Example

I) P->S
II) S->x;S
III) S->e

<u>Input string</u>

x;x;e



State		x	;	e	P	S	Action
		Symbol					
	0	1		3		5	Shift
	1		2				Shift
	2	1		3		4	Shift
	3						Reduce 3
	4						Reduce 2
	5						Accept

Step	Parse Stack	Rem. Input	Parser Action
1	0	`x;x;e`	Shift(1)
2	0 1	`;x;e`	Shift(2)
3	0 1 2	`x;e`	Shift(1)
4	0 1 2 1	`;e`	Shift(2)
5	0 1 2 1 2	e	Shift(3)
6	0 1 2 1 2 3		Reduce 3
7	0 1 2 1 2		

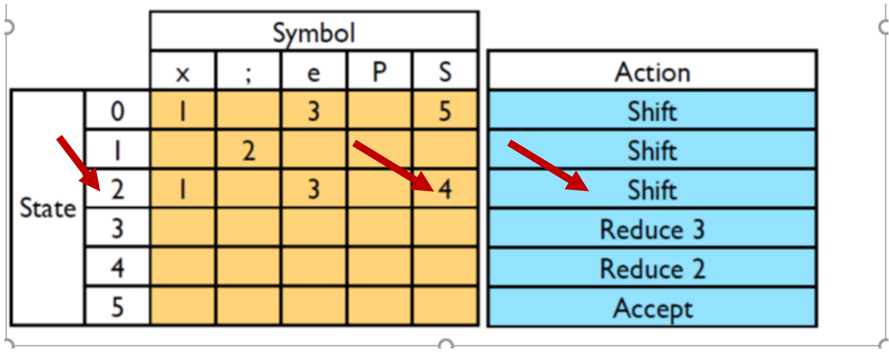
- *Now stack top has symbol 2 and LHS of rule III has S (imagine you saw S at input). Consult goto and action table.*

Example

I) P->S
II) S->x;S
III) S->e

<u>Input string</u>

`x;x;e`



		Symbol					
		x	;	e	P	S	Action
State	0	1		3		5	Shift
	1		2				Shift
	2	1		3		4	Shift
	3						Reduce 3
	4						Reduce 2
	5						Accept

Step	Parse Stack	Rem. Input	Parser Action
1	0	`x;x;e`	Shift(1)
2	0 1	`;x;e`	Shift(2)
3	0 1 2	`x;e`	Shift(1)
4	0 1 2 1	`;e`	Shift(2)
5	0 1 2 1 2	`e`	Shift(3)
6	0 1 2 1 2 3		Reduce 3 (shift(4))
7	0 1 2 1 2 4		

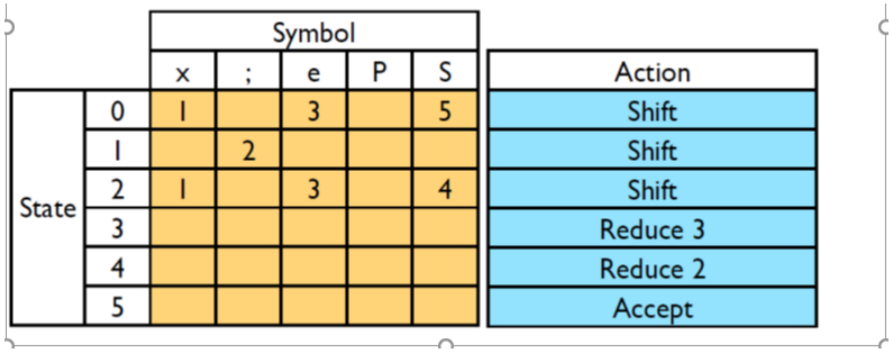
- *Now stack top has symbol 2 and LHS of rule III has S (imagine you saw S at input). Consult goto and action table. Shift(4)*

Example

I) P->S
II) S->x;S
III) S->e

<u>Input string</u>

x;x;e



		Symbol					
		x	;	e	P	S	Action
State	0	1		3		5	Shift
	1		2				Shift
	2	1		3		4	Shift
	3						Reduce 3
	4						Reduce 2
	5						Accept

Step	**Parse Stack**	**Rem. Input**	**Parser Action**
1	0	`x;x;e`	Shift(1)
2	0 1	`;x;e`	Shift(2)
3	0 1 2	`x;e`	Shift(1)
4	0 1 2 1	`;e`	Shift(2)
5	0 1 2 1 2	e	Shift(3)
6	0 1 2 1 2 3		Reduce 3 (shift(4))
7	0 1 2 1 2 4		**?**

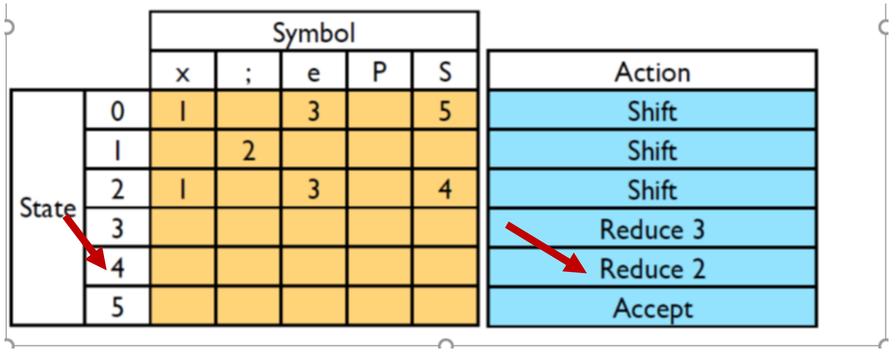
- *Now stack top has symbol 2 and LHS of rule III has S (imagine you saw S at input). Consult goto and action table. Shift(4)*

Example

I) P->S
II) S->x;S
III) S->e

<u>Input string</u>

x;x;e



		Symbol					Action
		x	;	e	P	S	
State	0	1		3		5	Shift
	1		2				Shift
	2	1		3		4	Shift
	3						Reduce 3
	4						Reduce 2
	5						Accept

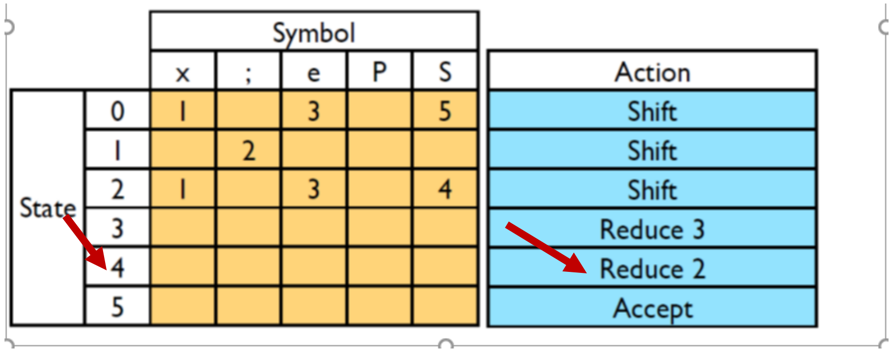
Step	**Parse Stack**	**Rem. Input**	**Parser Action**
1	0	x;x;e	Shift(1)
2	0 1	;x;e	Shift(2)
3	0 1 2	x;e	Shift(1)
4	0 1 2 1	;e	Shift(2)
5	0 1 2 1 2	e	Shift(3)
6	0 1 2 1 2 3		Reduce 3 (shift(4))
7	0 1 2 1 2 4		Reduce 2

Example

```
I) P->S
II) S->x;S
III) S->e
```

Input string

x;x;e



		Symbol					Action
		x	;	e	P	S	
State	0	1		3		5	Shift
	1		2				Shift
	2	1		3		4	Shift
	3						Reduce 3
	4						Reduce 2
	5						Accept

Step	**Parse Stack**	**Rem. Input**	**Parser Action**
1	0	`x;x;e`	Shift(1)
2	0 1	`;x;e`	Shift(2)
3	0 1 2	`x;e`	Shift(1)
4	0 1 2 1	`;e`	Shift(2)
5	0 1 2 1 2	`e`	Shift(3)
6	0 1 2 1 2 3		Reduce 3 (shift(4))
7	0 1 2 1 2 4		Reduce 2
8	0 1 2		

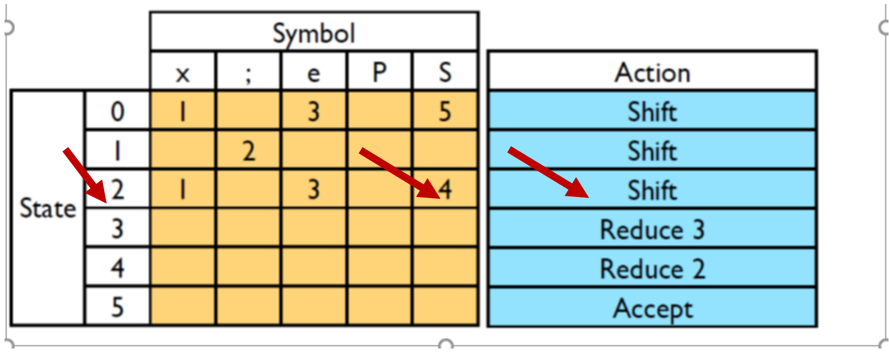
- *Look at rule II and pop 3 symbols of the stack because RHS of rule II has 3 symbols*

Example

I) P->S
II) S->x;S
III) S->e

<u>Input string</u>

x;x;e



		Symbol					
		x	;	e	P	S	Action
State	0	1		3		5	Shift
	1		2				Shift
	2	1		3		4	Shift
	3						Reduce 3
	4						Reduce 2
	5						Accept

Step	**Parse Stack**	**Rem. Input**	**Parser Action**
1	0	`x;x;e`	Shift(1)
2	0 1	`;x;e`	Shift(2)
3	0 1 2	`x;e`	Shift(1)
4	0 1 2 1	`;e`	Shift(2)
5	0 1 2 1 2	e	Shift(3)
6	0 1 2 1 2 3		Reduce 3 (shift(4))
7	0 1 2 1 2 4		Reduce 2
8	0 1 2		

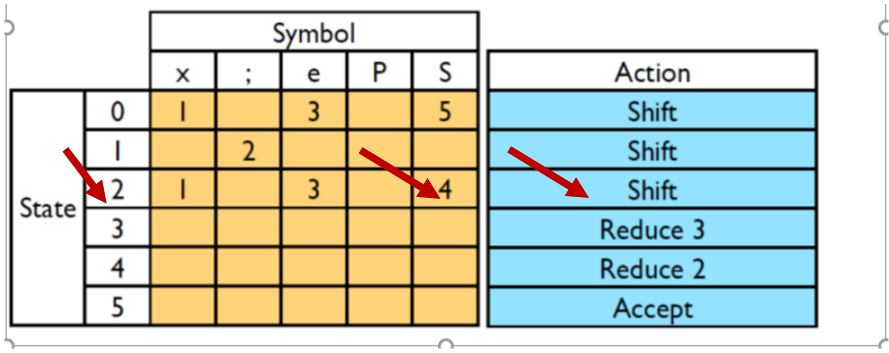
- *Now stack top has symbol 2 and LHS of rule II has S (imagine you saw S at input). Consult goto and action table.*

Example

```
I) P->S
II) S->x;S
III) S->e
```

<u>Input string</u>

x;x;e



	Symbol					
State	x	;	e	P	S	Action
0	I		3		5	Shift
I		2				Shift
2	I		3		4	Shift
3						Reduce 3
4						Reduce 2
5						Accept

Step	**Parse Stack**	**Rem. Input**	**Parser Action**
1	0	`x;x;e`	Shift(1)
2	0 1	`;x;e`	Shift(2)
3	0 1 2	`x;e`	Shift(1)
4	0 1 2 1	`;e`	Shift(2)
5	0 1 2 1 2	`e`	Shift(3)
6	0 1 2 1 2 3		Reduce 3 (shift(4))
7	0 1 2 1 2 4		Reduce 2 (shift(4))
8	0 1 2 4		

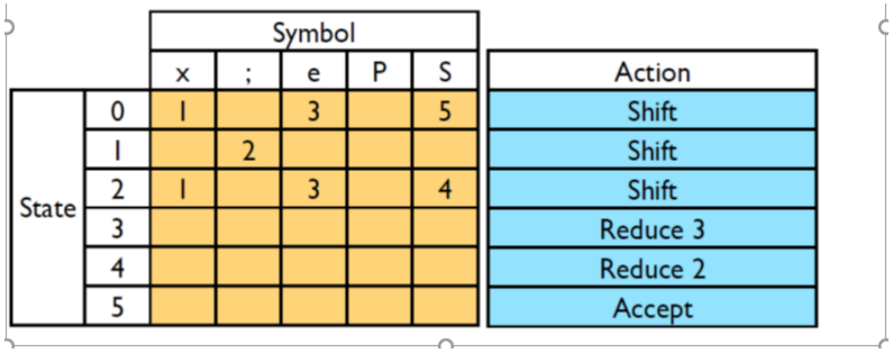
- *Now stack top has symbol 2 and LHS of rule II has S (imagine you saw S at input). Consult goto and action table. Shift(4)*

Example

```
I) P->S
II) S->x;S
III) S->e
```

Input string

x;x;e



		Symbol					Action
		x	;	e	P	S	
State	0	1		3		5	Shift
	1		2				Shift
	2	1		3		4	Shift
	3						Reduce 3
	4						Reduce 2
	5						Accept

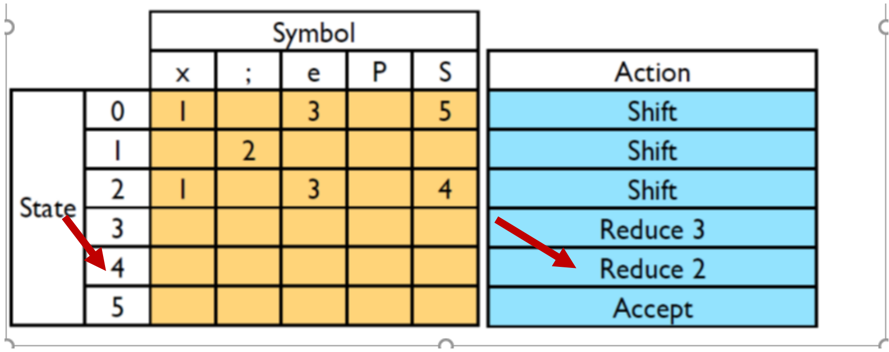
Step	**Parse Stack**	**Rem. Input**	**Parser Action**
1	0	x;x;e	Shift(1)
2	0 1	;x;e	Shift(2)
3	0 1 2	x;e	Shift(1)
4	0 1 2 1	;e	Shift(2)
5	0 1 2 1 2	e	Shift(3)
6	0 1 2 1 2 3		Reduce 3 (shift(4))
7	0 1 2 1 2 4		Reduce 2 (shift(4))
8	0 1 2 4		**?**

Example

```
I) P->S
II) S->x;S
III) S->e
```

<u>Input string</u>

`x;x;e`



		Symbol					Action
		x	;	e	P	S	
State	0	1		3		5	Shift
	1		2				Shift
	2	1		3		4	Shift
	3						Reduce 3
	4						Reduce 2
	5						Accept

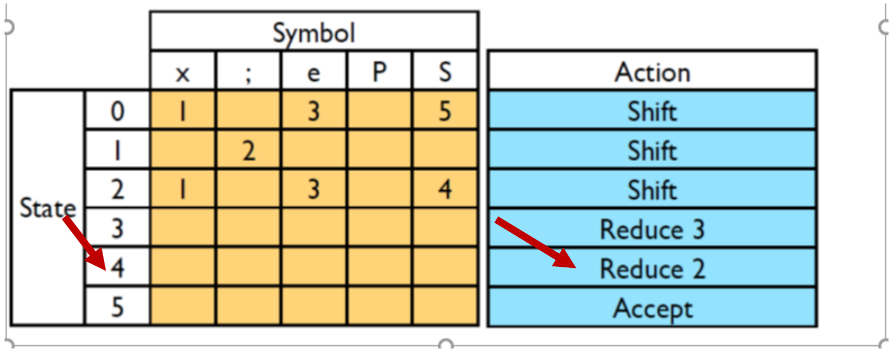
Step	Parse Stack	Rem. Input	Parser Action
1	0	`x;x;e`	Shift(1)
2	0 1	`;x;e`	Shift(2)
3	0 1 2	`x;e`	Shift(1)
4	0 1 2 1	`;e`	Shift(2)
5	0 1 2 1 2	e	Shift(3)
6	0 1 2 1 2 3		Reduce 3 (shift(4))
7	0 1 2 1 2 4		Reduce 2 (shift(4))
8	0 1 2 4		Reduce 2

Example

```
I) P->S
II) S->x;S
III) S->e
```

<u>Input string</u>

x;x;e



		x	;	e	P	S	Action
State	0	1		3		5	Shift
	1		2				Shift
	2	1		3		4	Shift
	3						Reduce 3
	4						Reduce 2
	5						Accept

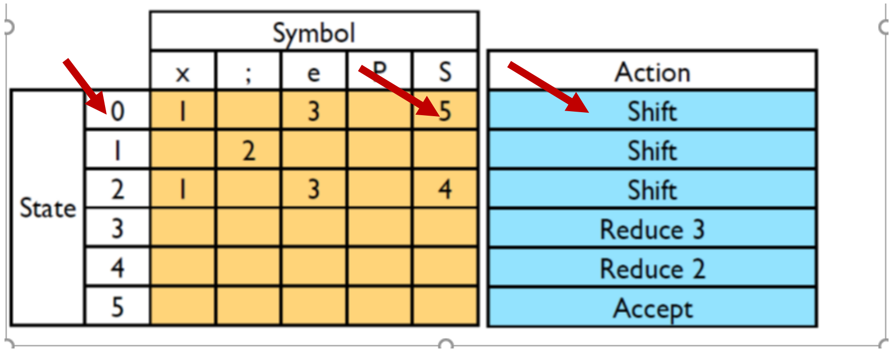
Step	**Parse Stack**	**Rem. Input**	**Parser Action**
1	0	`x;x;e`	Shift(1)
2	0 1	`;x;e`	Shift(2)
3	0 1 2	`x;e`	Shift(1)
4	0 1 2 1	`;e`	Shift(2)
5	0 1 2 1 2	`e`	Shift(3)
6	0 1 2 1 2 3		Reduce 3 (shift(4))
7	0 1 2 1 2 4		Reduce 2 (shift(4))
8	0 1 2 4		Reduce 2
9	0		

Example

```
I) P->S
II) S->x;S
III) S->e
```

Input string

`x;x;e`



State	x	;	e	P	S	Action
0	1		3		5	Shift
1		2				Shift
2	1		3		4	Shift
3						Reduce 3
4						Reduce 2
5						Accept

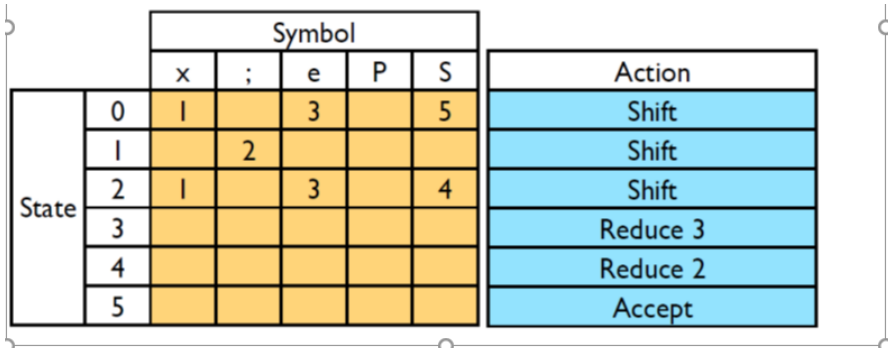
Step	Parse Stack	Rem. Input	Parser Action
1	0	x;x;e	Shift(1)
2	0 1	;x;e	Shift(2)
3	0 1 2	x;e	Shift(1)
4	0 1 2 1	;e	Shift(2)
5	0 1 2 1 2	e	Shift(3)
6	0 1 2 1 2 3		Reduce 3 (shift(4))
7	0 1 2 1 2 4		Reduce 2 (shift(4))
8	0 1 2 4		Reduce 2 (shift(5))
9	0 5		

Example

```
I) P->S
II) S->x;S
III) S->e
```

<u>Input string</u>

x;x;e



		Symbol					
		x	;	e	P	S	Action
State	0	1		3		5	Shift
	1		2				Shift
	2	1		3		4	Shift
	3						Reduce 3
	4						Reduce 2
	5						Accept

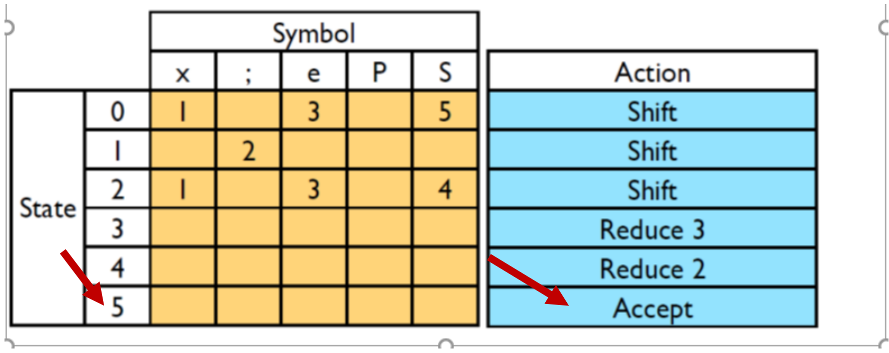
Step	**Parse Stack**	**Rem. Input**	**Parser Action**
1	0	x;x;e	Shift(1)
2	0 1	;x;e	Shift(2)
3	0 1 2	x;e	Shift(1)
4	0 1 2 1	;e	Shift(2)
5	0 1 2 1 2	e	Shift(3)
6	0 1 2 1 2 3		Reduce 3 (shift(4))
7	0 1 2 1 2 4		Reduce 2 (shift(4))
8	0 1 2 4		Reduce 2 (shift(5))
9	0 5		**?**

Example

I) P->S
II) S->x;S
III) S->e

<u>Input string</u>

x;x;e



		Symbol					
		x	;	e	P	S	Action
State	0	1		3		5	Shift
	1		2				Shift
	2	1		3		4	Shift
	3						Reduce 3
	4						Reduce 2
	5						Accept

Step	Parse Stack	Rem. Input	Parser Action
1	0	x;x;e	Shift(1)
2	0 1	;x;e	Shift(2)
3	0 1 2	x;e	Shift(1)
4	0 1 2 1	;e	Shift(2)
5	0 1 2 1 2	e	Shift(3)
6	0 1 2 1 2 3		Reduce 3 (shift(4))
7	0 1 2 1 2 4		Reduce 2 (shift(4))
8	0 1 2 4		Reduce 2 (shift(5))
9	0 5		Accept

means replace whatever is there in the stack with the start symbol

Example

```
I) P->S
II) S->x;S
III) S->e
```

Input string

|x;x;e

Initial scan pointer

Step	Parser Action
1	Shift(1)
2	Shift(2)
3	Shift(1)
4	Shift(2)
5	Shift(3)
6	Reduce 3 (shift(4))
7	Reduce 2 (shift(4))
8	Reduce 2 (shift(5))
9	Accept

Example

```
I) P->S
II) S->x;S
III) S->e
```

Input string

|x;x;e

Initial scan pointer

Step	Parser Action
1	Shift(1)
2	Shift(2)
3	Shift(1)
4	Shift(2)
5	Shift(3)
6	Reduce 3 (shift(4))
7	Reduce 2 (shift(4))
8	Reduce 2 (shift(5))
9	Accept

x |; x ; e

Example

```
I) P->S
II) S->x;S
III) S->e
```

Input string

|x;x;e

Initial scan pointer

Step	Parser Action
1	Shift(1)
2	Shift(2)
3	Shift(1)
4	Shift(2)
5	Shift(3)
6	Reduce 3 (shift(4))
7	Reduce 2 (shift(4))
8	Reduce 2 (shift(5))
9	Accept

x ;|x ; e

Example

```
I) P->S
II) S->x;S
III) S->e
```

Input string

|x;x;e

Initial scan pointer

Step	Parser Action
1	Shift(1)
2	Shift(2)
3	Shift(1)
4	Shift(2)
5	Shift(3)
6	Reduce 3 (shift(4))
7	Reduce 2 (shift(4))
8	Reduce 2 (shift(5))
9	Accept

x ; x|; e

Example

```
I) P->S
II) S->x;S
III) S->e
```

Input string

|x;x;e

Initial scan pointer

Step	Parser Action
1	Shift(1)
2	Shift(2)
3	Shift(1)
4	Shift(2)
5	Shift(3)
6	Reduce 3 (shift(4))
7	Reduce 2 (shift(4))
8	Reduce 2 (shift(5))
9	Accept

x ; x ;| e

Example

```
I) P->S
II) S->x;S
III) S->e
```

Input string

|x;x;e

Initial scan pointer

Step	Parser Action
1	Shift(1)
2	Shift(2)
3	Shift(1)
4	Shift(2)
5	Shift(3)
6	Reduce 3 (shift(4))
7	Reduce 2 (shift(4))
8	Reduce 2 (shift(5))
9	Accept

x ; x ; e|

Example

I) P->S
II) S->x;S
III) S->e

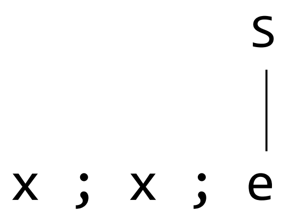
<u>Input string</u>

|x;x;e

Initial scan pointer

Step	Parser Action
1	Shift(1)
2	Shift(2)
3	Shift(1)
4	Shift(2)
5	Shift(3)
6	Reduce 3 (shift(4))
7	Reduce 2 (shift(4))
8	Reduce 2 (shift(5))
9	Accept

x ; x ; e|



Example

```
I) P->S
II) S->x;S
III) S->e
```

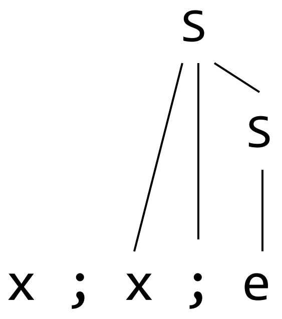
Input string

|x;x;e

Initial scan pointer

Step	Parser Action
1	Shift(1)
2	Shift(2)
3	Shift(1)
4	Shift(2)
5	Shift(3)
6	Reduce 3 (shift(4))
7	Reduce 2 (shift(4))
8	Reduce 2 (shift(5))
9	Accept

x ; x ; S|



Example

I) P->S
II) S->x;S
III) S->e

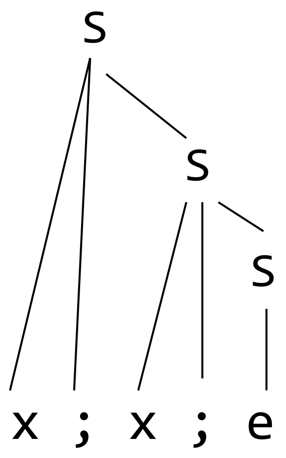
Input string

|x;x;e

Initial scan pointer

Step	Parser Action
1	Shift(1)
2	Shift(2)
3	Shift(1)
4	Shift(2)
5	Shift(3)
6	Reduce 3 (shift(4))
7	Reduce 2 (shift(4))
8	Reduce 2 (shift(5))
9	Accept

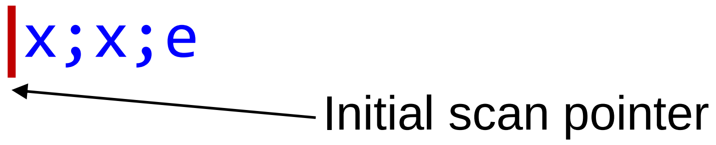
x ; S |



Example

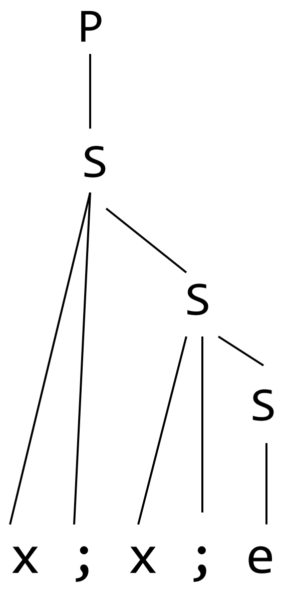
```
I) P->S
II) S->x;S
III) S->e
```

Input string



Step	Parser Action
1	Shift(1)
2	Shift(2)
3	Shift(1)
4	Shift(2)
5	Shift(3)
6	Reduce 3 (shift(4))
7	Reduce 2 (shift(4))
8	Reduce 2 (shift(5))
9	Accept

S |



Parsing using an LR(0) parser

- Basic idea: parser keeps track, simultaneously, of all possible productions that *could be matched* given what it's seen so far. When it sees a full production, match it.
- Maintain a *parse stack* that tells you what state you're in
  - Start in state 0
- In each state, look up in action table whether to:
  - *shift*: consume a token off the input; look for next state in goto table; push next state onto stack
  - *reduce*: match a production; pop off as many symbols from state stack as seen in production; look up where to go according to non-terminal we just matched; push next state onto stack
  - *accept*: terminate parse

Shift-Reduce Parsing

The LR parsing seen previously is an example of shift-reduce parsing

- When do we *shift* and when do we *reduce*?
  - *How do we construct goto and action tables?*