Parsing VI
The Last Parsing Lecture

Example

Simple left recursive SheepNoise grammar

Note: Example in book is right recursive and generates different Action and Goto tables

Goal → *SheepNoise*

SheepNoise → *SheepNoise baa*

SheepNoise→ *baa*

Example From SheepNoise

Initial step builds the item [*Goal*→•*SheepNoise*,EOF] and takes its *closure( )*

Closure([*Goal*→•*SheepNoise*,EOF] *)*

Item
[*Goal*→ •*SheepNoise*, EOF]
[*SheepNoise*→ •*SheepNoise* baa, EOF]
[*SheepNoise*→ • baa, EOF]
[*SheepNoise*→ •*SheepNoise* baa, baa]
[*SheepNoise*→ • baa, baa]

So, S_0 is

{ [*Goal*→ • *SheepNoise*, EOF], [*SheepNoise*→ • *SheepNoise* baa, EOF], [*SheepNoise*→• baa, EOF], [*SheepNoise*→ • *SheepNoise* baa, baa], [*SheepNoise*→ • baa, baa] }

Example from SheepNoise

S_0 is { [*Goal*→ • *SheepNoise*,EOF], [*SheepNoise*→ • *SheepNoise* baa,EOF],
[*SheepNoise*→ • baa,EOF], [*SheepNoise*→ • *SheepNoise* baa,baa],
[*SheepNoise*→ • baa,baa] }

Goto(S_0 , baa)

- Loop produces

Item	*From*
[*SheepNoise*→baa•, EOF]	Item 3 in s_0
[*SheepNoise*→baa•, baa]	Item 5 in s_0

- Closure adds nothing since • is at end of *rhs* in each item

In the construction, this produces s_2

{ [*SheepNoise*→baa •, {EOF,baa}]}

New, but *obvious*, notation for two distinct items
[*SheepNoise*→baa •, EOF] &
[*SheepNoise*→baa •, baa]

Example from SheepNoise

Starts with S_0

S_0 : { [***Goal***→ • ***SheepNoise***, EOF], [***SheepNoise***→ • ***SheepNoise*** baa, EOF],
[***SheepNoise***→• baa, EOF], [***SheepNoise***→ • ***SheepNoise*** baa, baa],
[***SheepNoise***→ • baa, baa] }

Example from SheepNoise

Starts with S_0

S_0 : { [*Goal*→ • ***SheepNoise***, EOF], [***SheepNoise***→ • ***SheepNoise*** baa, EOF],
[***SheepNoise***→• baa, EOF], [***SheepNoise***→ • ***SheepNoise*** baa, baa],
[***SheepNoise***→ • baa, baa] }

Iteration 1 computes

S_1 = *Goto*(S_0 , ***SheepNoise***) =
{ [*Goal*→ ***SheepNoise*** •, EOF], [***SheepNoise***→ ***SheepNoise*** • baa, EOF],
[***SheepNoise***→ ***SheepNoise*** • baa, baa] }

S_2 = *Goto*(S_0 , baa) = { [***SheepNoise***→ baa •, EOF],
[***SheepNoise***→ baa •, baa] }

Example from SheepNoise

Starts with S_0

S_0 : { [*Goal*→ • ***SheepNoise***, EOF], [***SheepNoise***→ • ***SheepNoise*** baa, EOF],
[***SheepNoise***→• baa, EOF], [***SheepNoise***→ • ***SheepNoise*** baa, baa],
[***SheepNoise***→ • baa, baa] }

Iteration 1 computes

S_1 = *Goto*(S_0 , ***SheepNoise***) =
{ [*Goal*→ ***SheepNoise*** •, EOF], [***SheepNoise***→ ***SheepNoise*** • baa, EOF],
[***SheepNoise***→ ***SheepNoise*** • baa, baa] }

S_2 = *Goto*(S_0 , baa) = { [***SheepNoise***→ baa •, EOF],
[***SheepNoise***→ baa •, baa] }

Iteration 2 computes

S_3 = *Goto*(S_1 , baa) = { [***SheepNoise***→ ***SheepNoise*** baa •, EOF],
[***SheepNoise***→ ***SheepNoise*** baa •, baa] }

Example from SheepNoise

Starts with S_0

S_0 : { [*Goal*→ • ***SheepNoise***, EOF], [***SheepNoise***→ • ***SheepNoise*** baa, EOF],
[***SheepNoise***→• baa, EOF], [***SheepNoise***→ • ***SheepNoise*** baa, baa],
[***SheepNoise***→ • baa, baa] }

Iteration 1 computes

S_1 = *Goto*(S_0 , ***SheepNoise***) =
{ [*Goal*→ ***SheepNoise*** •, EOF], [***SheepNoise***→ ***SheepNoise*** • baa, EOF],
[***SheepNoise***→ ***SheepNoise*** • baa, baa] }

S_2 = *Goto*(S_0 , baa) = { [***SheepNoise***→ baa •, EOF],
[***SheepNoise***→ baa •, baa] }

Iteration 2 computes

S_3 = *Goto*(S_1 , baa) = { [***SheepNoise***→ ***SheepNoise*** baa •, EOF],
[***SheepNoise***→ ***SheepNoise*** baa •, baa] }

Nothing more to compute, since • is at the end of every item in S_3.

Example from SheepNoise

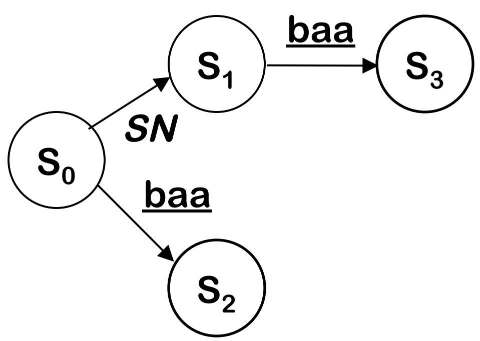
S_0 : { [***Goal***→ • ***SheepNoise***, EOF], [***SheepNoise***→ • ***SheepNoise*** baa, EOF],
[***SheepNoise***→• baa, EOF], [***SheepNoise***→ • ***SheepNoise*** baa, baa],
[***SheepNoise***→ • baa, baa] }

S_1 = *Goto*(S_0 , ***SheepNoise***) =
{ [***Goal***→ ***SheepNoise*** •, EOF], [***SheepNoise***→ ***SheepNoise*** • baa, EOF],
[***SheepNoise***→ ***SheepNoise*** • baa, baa] }

S_2 = *Goto*(S_0 , baa) = { [***SheepNoise***→ baa •, EOF],
[***SheepNoise***→ baa •, baa] }

S_3 = *Goto*(S_1 , baa) = { [***SheepNoise***→ ***SheepNoise*** baa •, EOF],
[***SheepNoise***→ ***SheepNoise*** baa •, baa] }

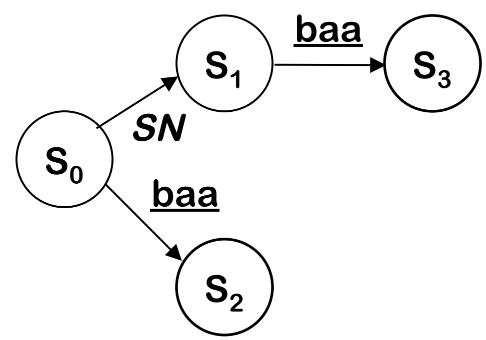
Control DFA for SN



Filling in the ACTION and GOTO Tables

$\forall$ *set* $s_x \in S$
- $\forall$ *item* $i \in s_x$
  - *if* i *is* $[A \rightarrow \beta \cdot \underline{a}d, \underline{b}]$ *and* $goto(s_x, \underline{a}) = s_k$, $\underline{a} \in T$
    - *then* ACTION[x,$\underline{a}$] $\leftarrow$ "*shift k*"
  - *else if* i *is* $[S' \rightarrow S \cdot, \text{EOF}]$
    - *then* ACTION[x ,$\underline{a}$] $\leftarrow$ "*accept*"
  - *else if* i *is* $[A \rightarrow \beta \cdot, \underline{a}]$
    - *then* ACTION[x,$\underline{a}$] $\leftarrow$ "*reduce* $A \rightarrow \beta$"
- $\forall$ $n \in NT$
  - *if* $goto(s_x ,n) = s_k$
    - *then* GOTO[x,n] $\leftarrow$ k

Control DFA for SN



S_0 : { [*Goal*→ · *SheepNoise*, EOF], [*SheepNoise*→ · *SheepNoise* baa, EOF],
[*SheepNoise*→ · baa, EOF], [*SheepNoise*→ · *SheepNoise* baa, baa],
[*SheepNoise*→ · baa, baa] }

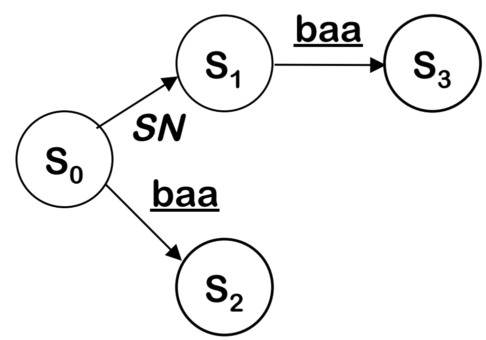
S_1 = *Goto*(S_0 , *SheepNoise*) =
{ [*Goal*→ *SheepNoise* ·, EOF], [*SheepNoise*→ *SheepNoise* · baa, EOF],
[*SheepNoise*→ *SheepNoise* · baa, baa] }

S_2 = *Goto*(S_0 , baa) = { [*SheepNoise*→ baa ·, EOF], [*SheepNoise*→ baa ·, baa] }

Filling in the ACTION and GOTO Tables

$\forall$ *set* $s_x \in S$
  $\forall$ *item* $i \in s_x$
    if i *is* $[A \rightarrow \beta \bullet \underline{a}d, \underline{b}]$ *and* $goto(s_x, \underline{a}) = s_k$, $\underline{a} \in T$
      then ACTION$[x, \underline{a}] \leftarrow$ "*shift k*"
    else if i *is* $[S' \rightarrow S \bullet, EOF]$
      then ACTION$[x, \underline{a}] \leftarrow$ "*accept*"
    else if i *is* $[A \rightarrow \beta \bullet, \underline{a}]$
      then ACTION$[x, \underline{a}] \leftarrow$ "*reduce* $A \rightarrow \beta$"
  $\forall$ $n \in NT$
    if $goto(s_x, n) = s_k$
      then GOTO$[x, n] \leftarrow k$

Control DFA for SN



$S_1 = Goto(S_0$, ***SheepNoise***$) =$
  { [***Goal***$\rightarrow$ ***SheepNoise*** $\bullet$, $\underline{EOF}$], [***SheepNoise***$\rightarrow$ ***SheepNoise*** $\bullet$ $\underline{baa}$, $\underline{EOF}$],
  [***SheepNoise***$\rightarrow$ ***SheepNoise*** $\bullet$ $\underline{baa}$, $\underline{baa}$] }

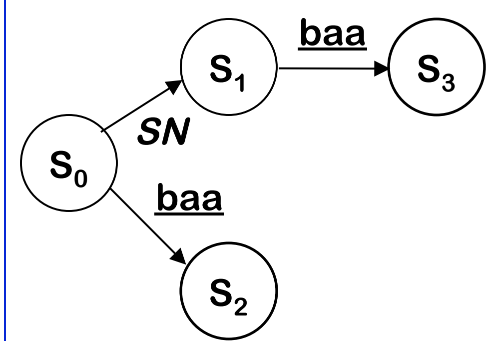
$S_2 = Goto(S_0, \underline{baa})$ = { [***SheepNoise***$\rightarrow$ $\underline{baa}$ $\bullet$, $\underline{EOF}$], [***SheepNoise***$\rightarrow$ $\underline{baa}$ $\bullet$, $\underline{baa}$] }

$S_3 = Goto(S_1, \underline{baa})$ = { [***SheepNoise***$\rightarrow$ ***SheepNoise*** $\underline{baa}$ $\bullet$, $\underline{EOF}$],
  [***SheepNoise***$\rightarrow$ ***SheepNoise*** $\underline{baa}$ $\bullet$, $\underline{baa}$] }

Filling in the ACTION and GOTO Tables

$\forall$ *set* $s_x \in S$
    $\forall$ *item* $i \in s_x$
        if i *is* $[A \rightarrow \beta \bullet \underline{a}\delta, \underline{b}]$ *and* $goto(s_x, \underline{a}) = s_k$, $\underline{a} \in T$
            then ACTION$[x,\underline{a}] \leftarrow$ "*shift k*"
        else if i *is* $[S' \rightarrow S \bullet, \text{EOF}]$
            then ACTION$[x, \underline{a}] \leftarrow$ "*accept*"
        else if i *is* $[A \rightarrow \beta \bullet, \underline{a}]$
            then ACTION$[x,\underline{a}] \leftarrow$ "*reduce* $A \rightarrow \beta$"
    $\forall$ $n \in NT$
        if $goto(s_x, n) = s_k$
            then GOTO$[x,n] \leftarrow k$

Control DFA for SN



ACTION		
State	EOF	baa
0	—	shift 2
1	accept	shift 3
2	reduce 3	reduce 3
3	reduce 2	reduce 2

GOTO	
State	*SheepNoise*
0	1
1	-
2	-
3	-

Shrinking the Tables

Three options:

- Combine terminals such as number & identifier, + & -, * & /
  - Directly removes a column, may remove a row
  - For expression grammar, 198 (vs. 384) table entries
- Combine rows or columns (*table compression*)
  - Implement identical rows once & remap states
  - Requires extra indirection on each lookup
  - Use separate mapping for ACTION & for GOTO
- Use another construction algorithm
  - Both LALR(1) and SLR(1) produce smaller tables
  - Implementations are readily available

What can go wrong with LR table construction?

What if set *s* contains [$A \rightarrow \beta \bullet \underline{a}\gamma, \underline{b}$] and [$B \rightarrow \beta \bullet, \underline{a}$] ?

- First item generates "shift", second generates "reduce"
- Both define ACTION[s,$\underline{a}$] — cannot do both actions
- This is a fundamental ambiguity, called a *shift/reduce error*
- Modify the grammar to eliminate it *(if-then-else)*
- Shifting will often resolve it correctly

EaC includes a worked example

What if set *s* contains [$A \rightarrow \gamma \bullet, \underline{a}$] and [$B \rightarrow \gamma \bullet, \underline{a}$] ?

- Each generates "reduce", but with a different production
- Both define ACTION[s,$\underline{a}$] — cannot do both reductions
- This fundamental ambiguity is called a *reduce/reduce error*
- Modify the grammar to eliminate it

In either case, the grammar is not LR(1)

Summary of top-down and LR(1) parsing

- Top down recursive descent parser
  - Advantages: Fast, good locality, simplicty
  - Disadvantages: Hand-coded, high maintenance

- LR(1) Parser
  - Advantages: Automatable
  - Disadvantages: Large working sets, large tables

CYK Parser

- Simple context-free-language parser
  - Worse-case running time is $O(n^3)$, space is $O(n^2)$
  - Employs bottom-up parsing and dynamic programming

- Shunned for many years

  "Even tabular methods [CYK, Earley] should be avoided if the language at hand has a grammar for which more efficient algorithms [LL, LALR] are available." The Theory of Parsing …., Aho, Ullman, 1972

- But in practice, running time is more like $O(n^{\approx 1.2})$
  - Plus computers are now 1,000,000-times faster than in 1972
  - And (more importantly) CYK parser is easily parallelizable!

Source: Ras Bodik, Slides: Browsing Web 3.0 on 3.0 Watts

CYK Parser (Sequential Version)

b	a	a	b	a

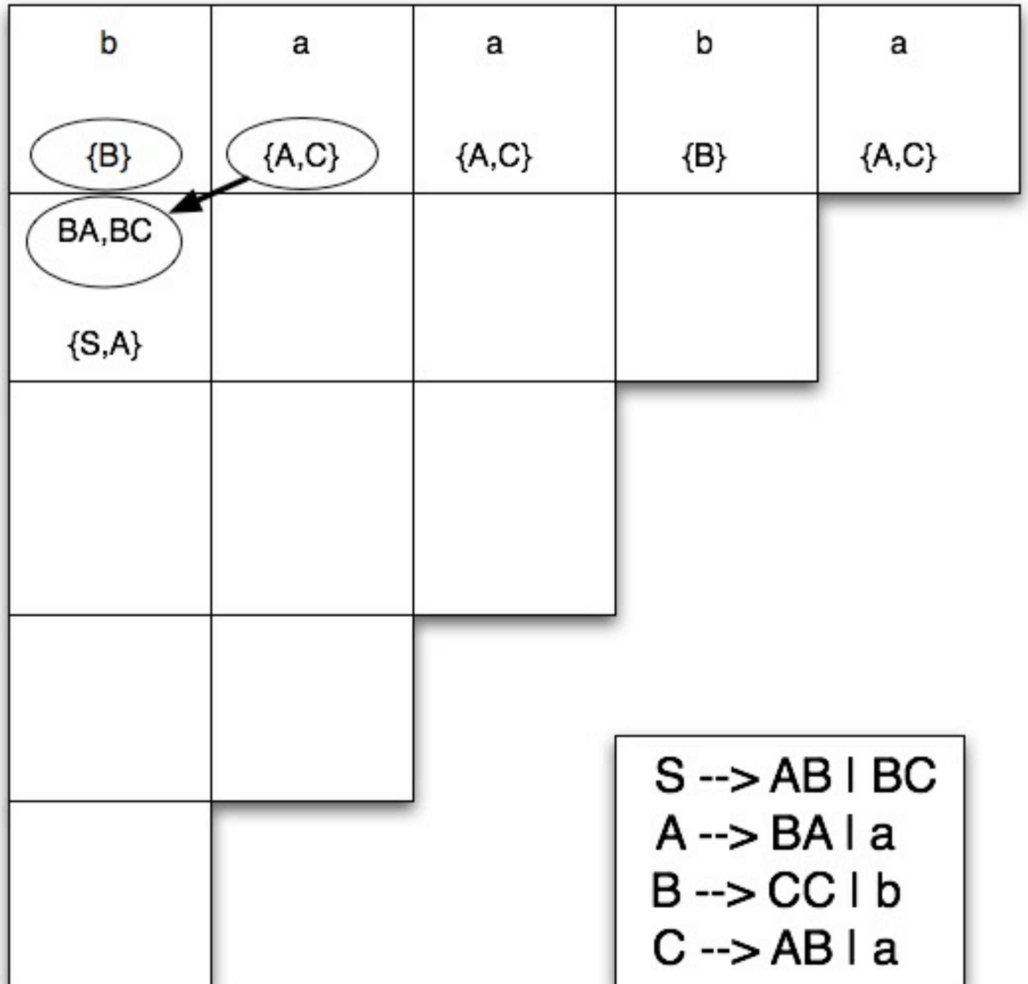
S --> AB | BC
A --> BA | a
B --> CC | b
C --> AB | a

CYK Parser (Sequential Version)

b	a	a	b	a
{B}	{A,C}	{A,C}	{B}	{A,C}

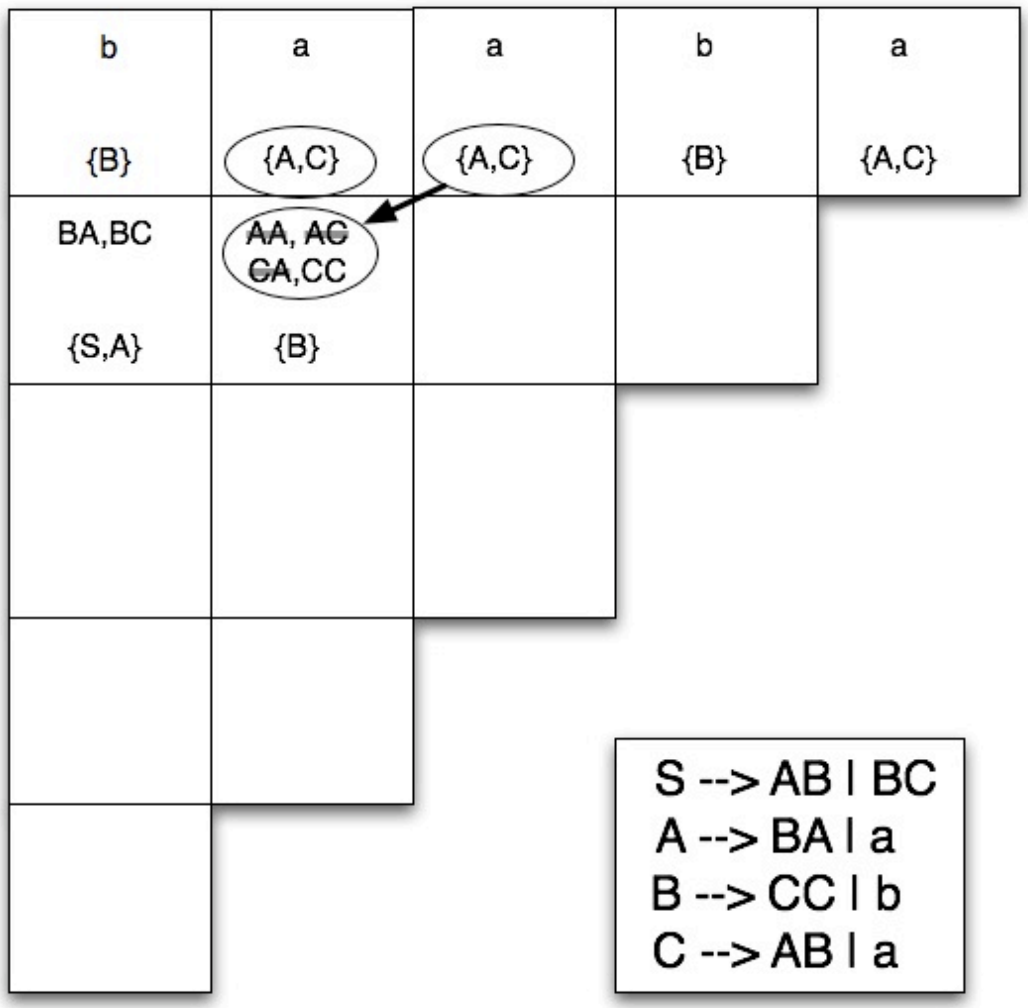
S --> AB | BC
A --> BA | a
B --> CC | b
C --> AB | a

CYK Parser (Sequential Version)

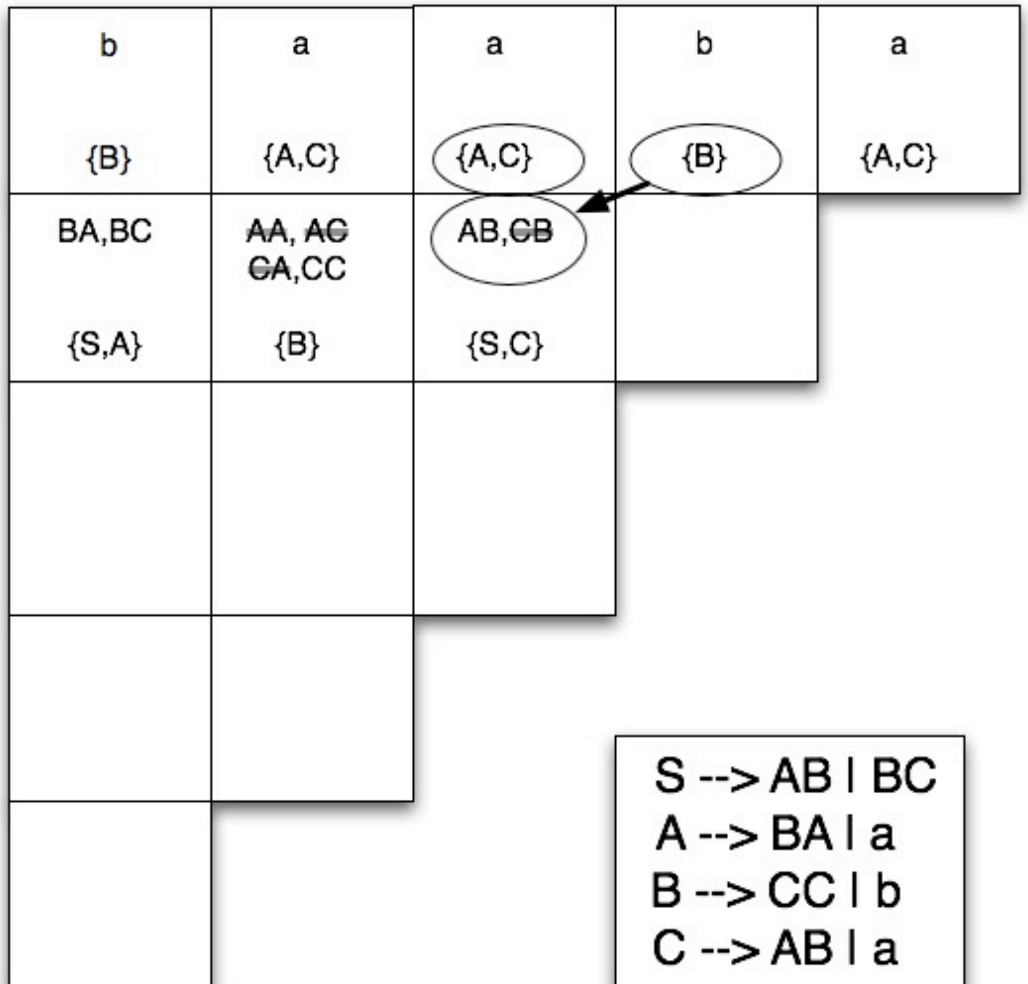


S --> AB | BC
A --> BA | a
B --> CC | b
C --> AB | a

CYK Parser (Sequential Version)

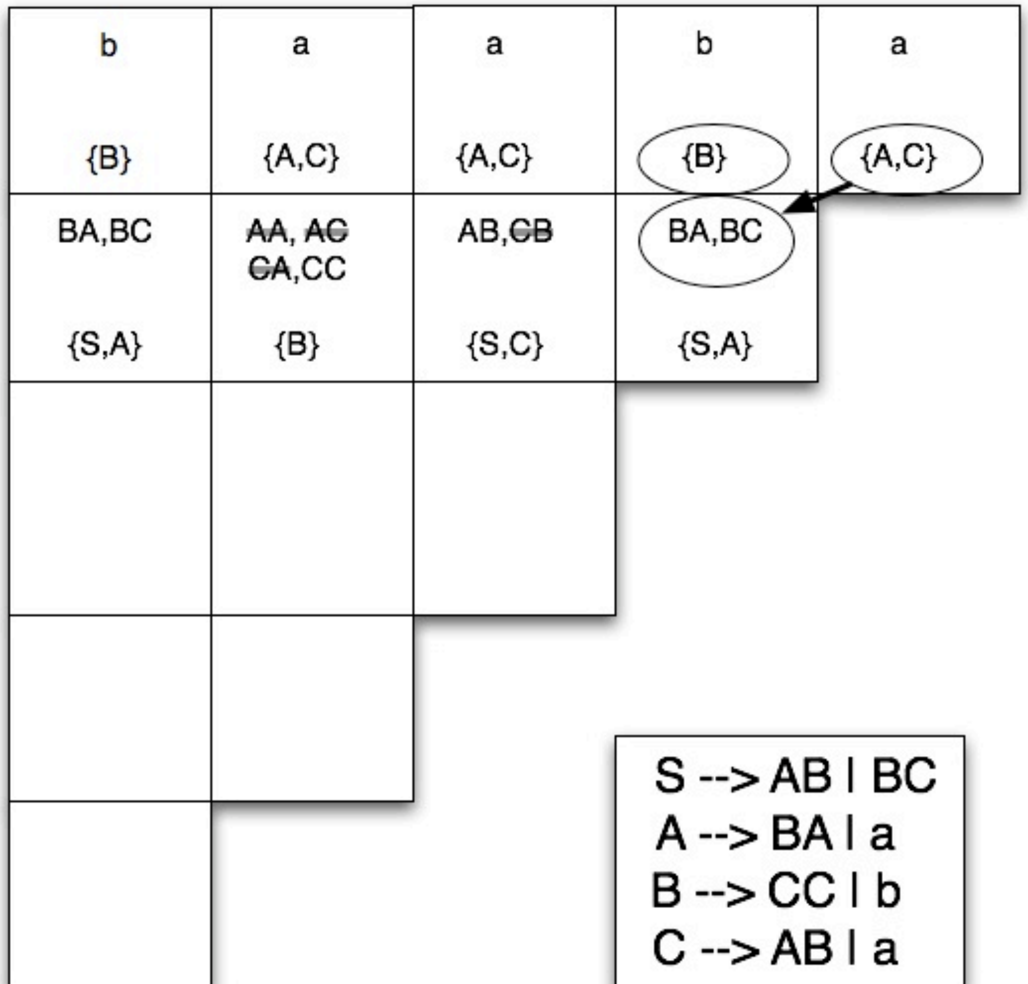


CYK Parser (Sequential Version)



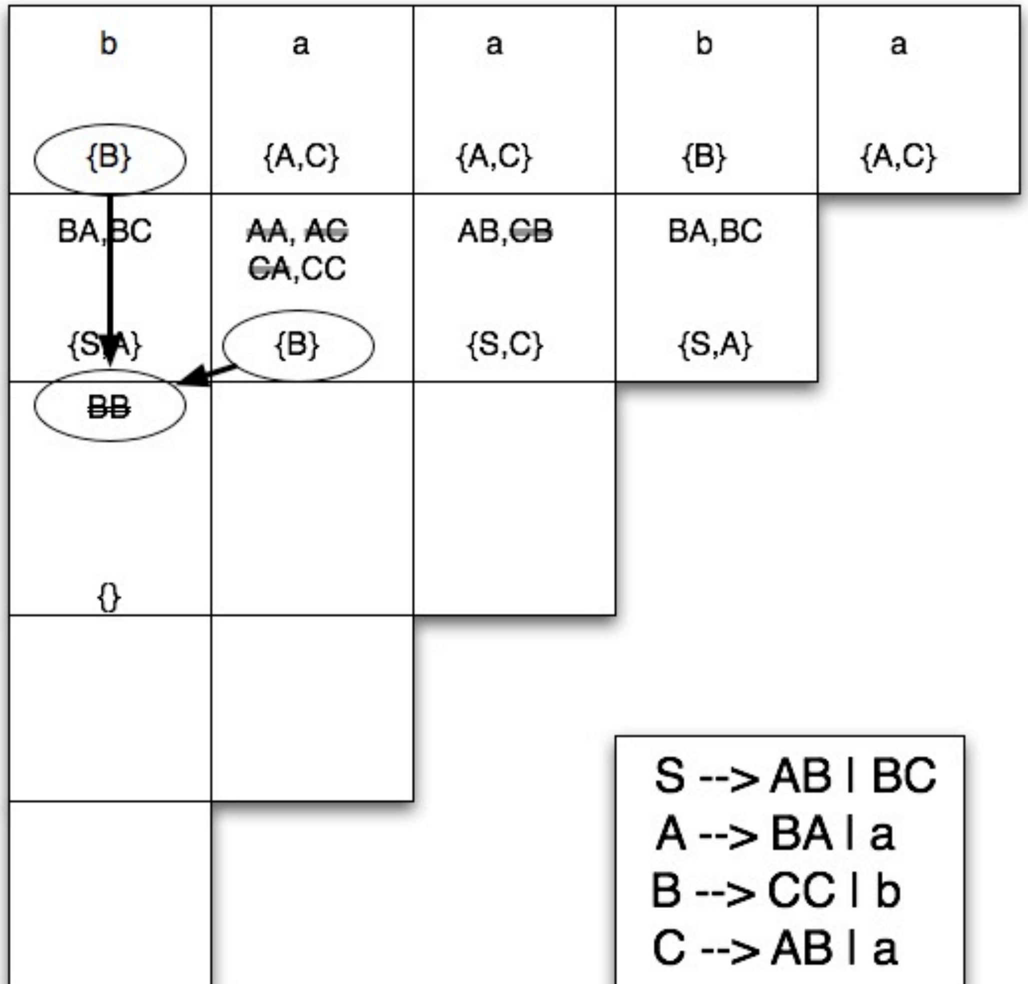
S --> AB | BC
A --> BA | a
B --> CC | b
C --> AB | a

CYK Parser (Sequential Version)



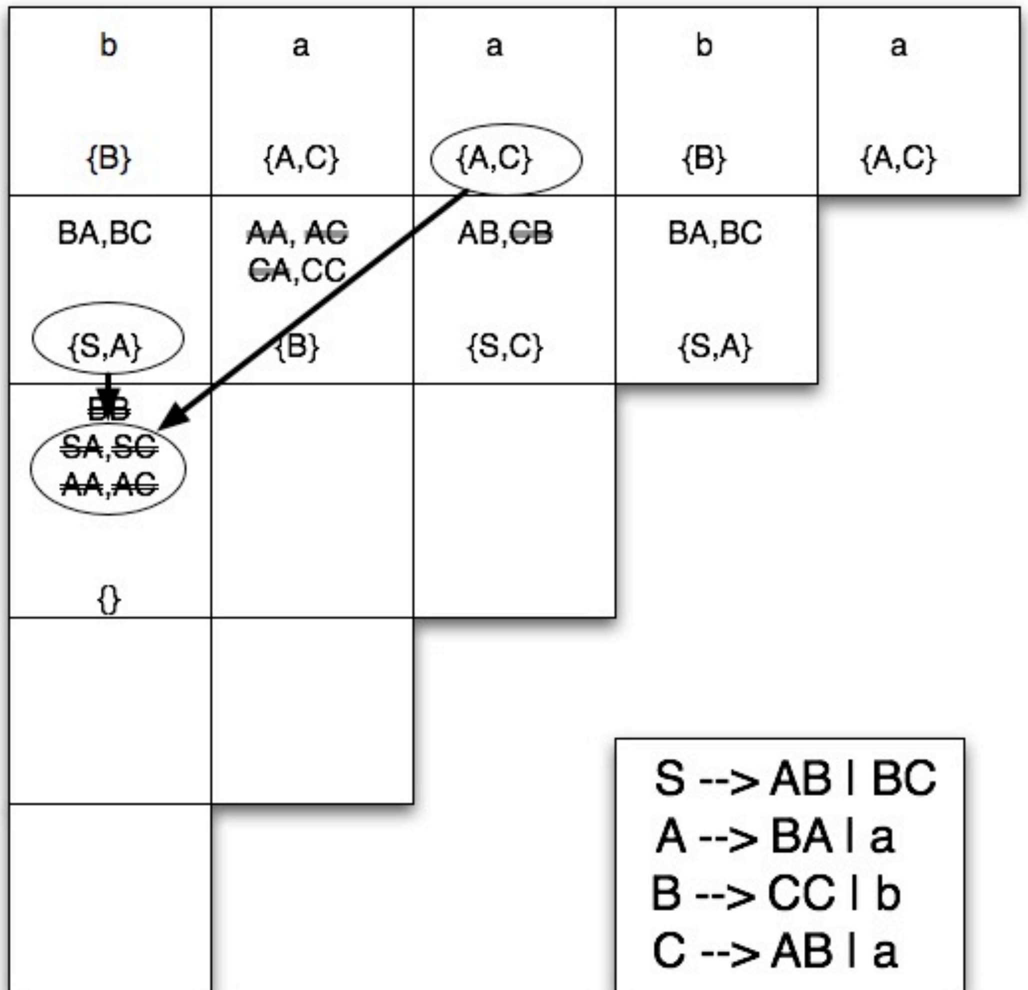
S --> AB | BC
A --> BA | a
B --> CC | b
C --> AB | a

CYK Parser (Sequential Version)



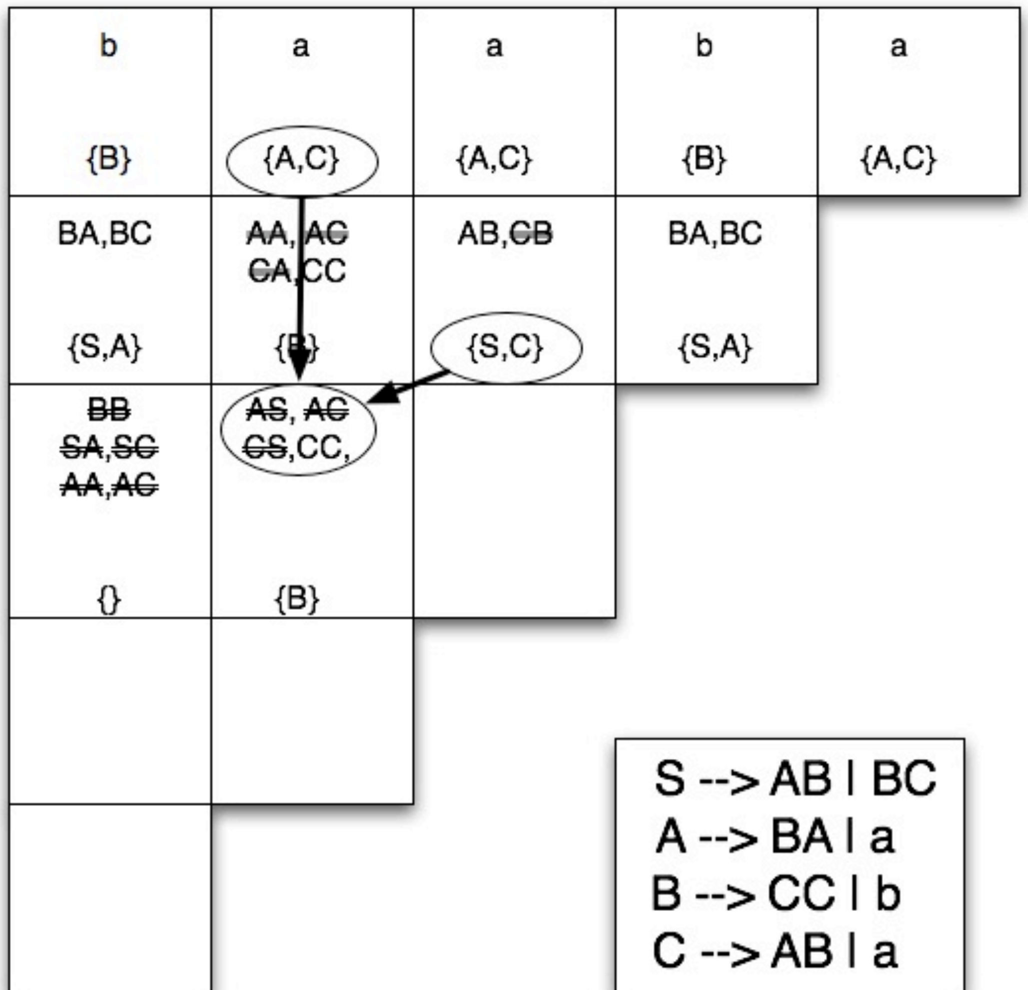
S --> AB | BC
A --> BA | a
B --> CC | b
C --> AB | a

CYK Parser (Sequential Version)



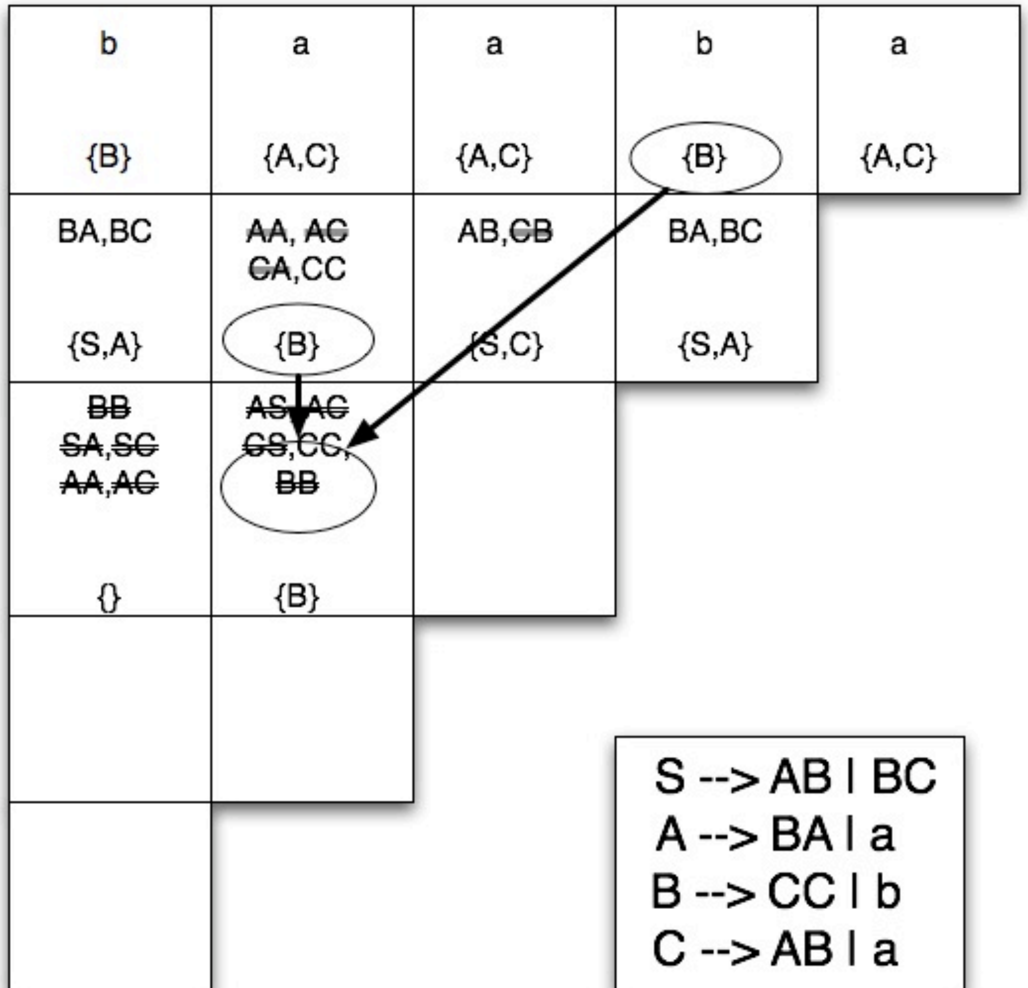
S --> AB | BC
A --> BA | a
B --> CC | b
C --> AB | a

CYK Parser (Sequential Version)



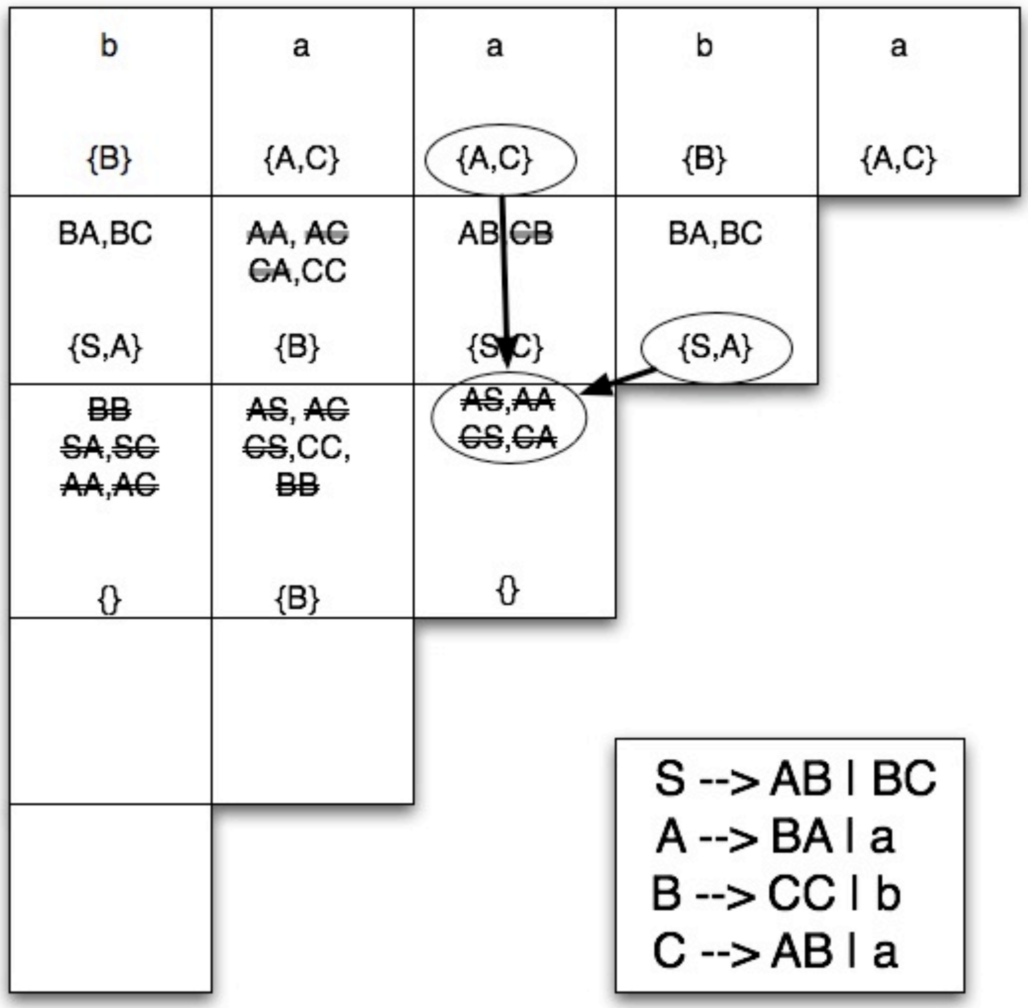
S --> AB | BC
A --> BA | a
B --> CC | b
C --> AB | a

CYK Parser (Sequential Version)

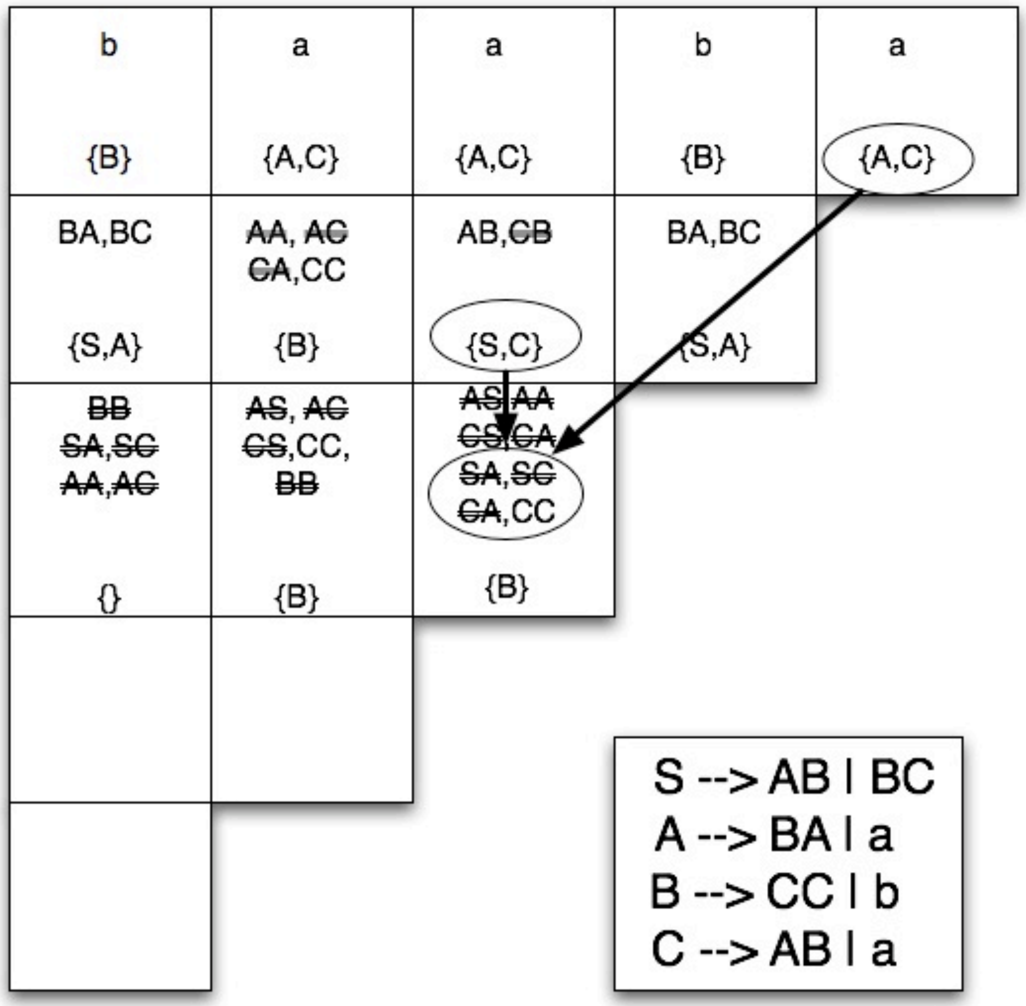


S --> AB | BC
A --> BA | a
B --> CC | b
C --> AB | a

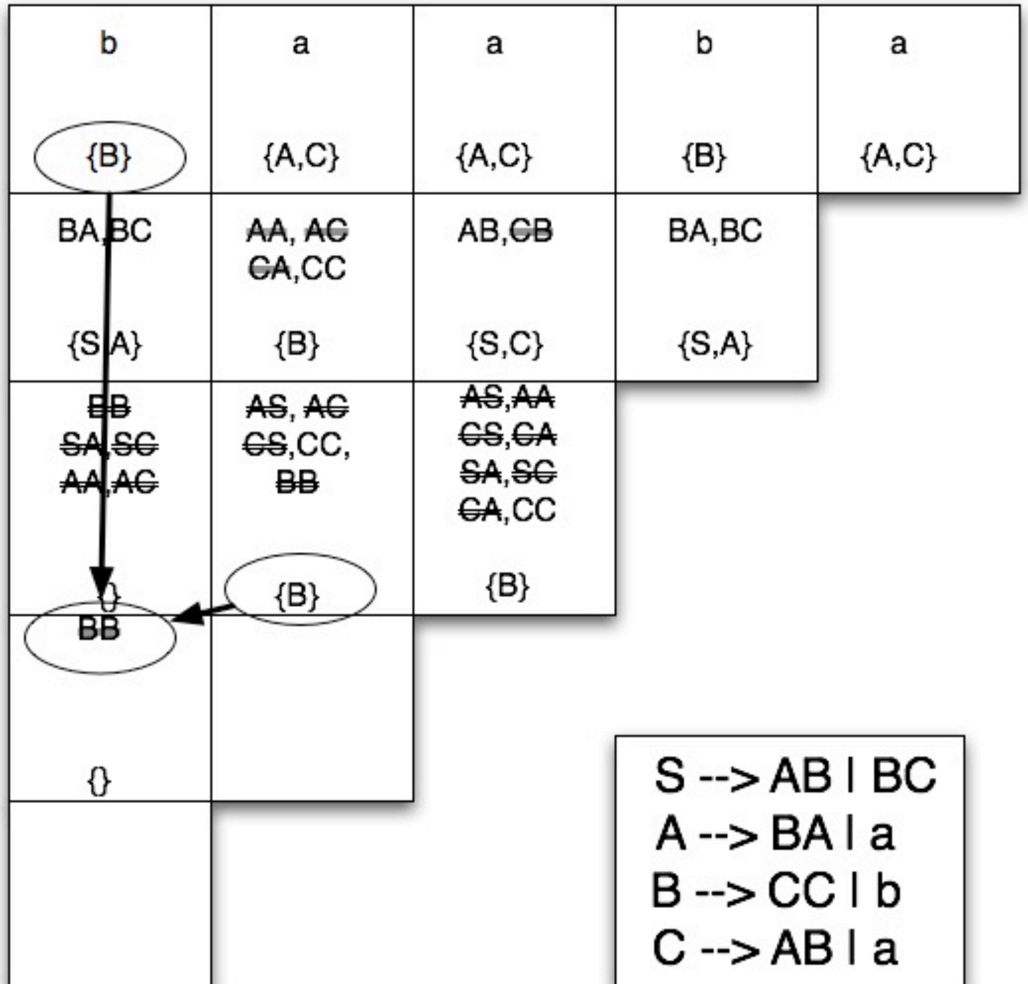
CYK Parser (Sequential Version)



CYK Parser (Sequential Version)

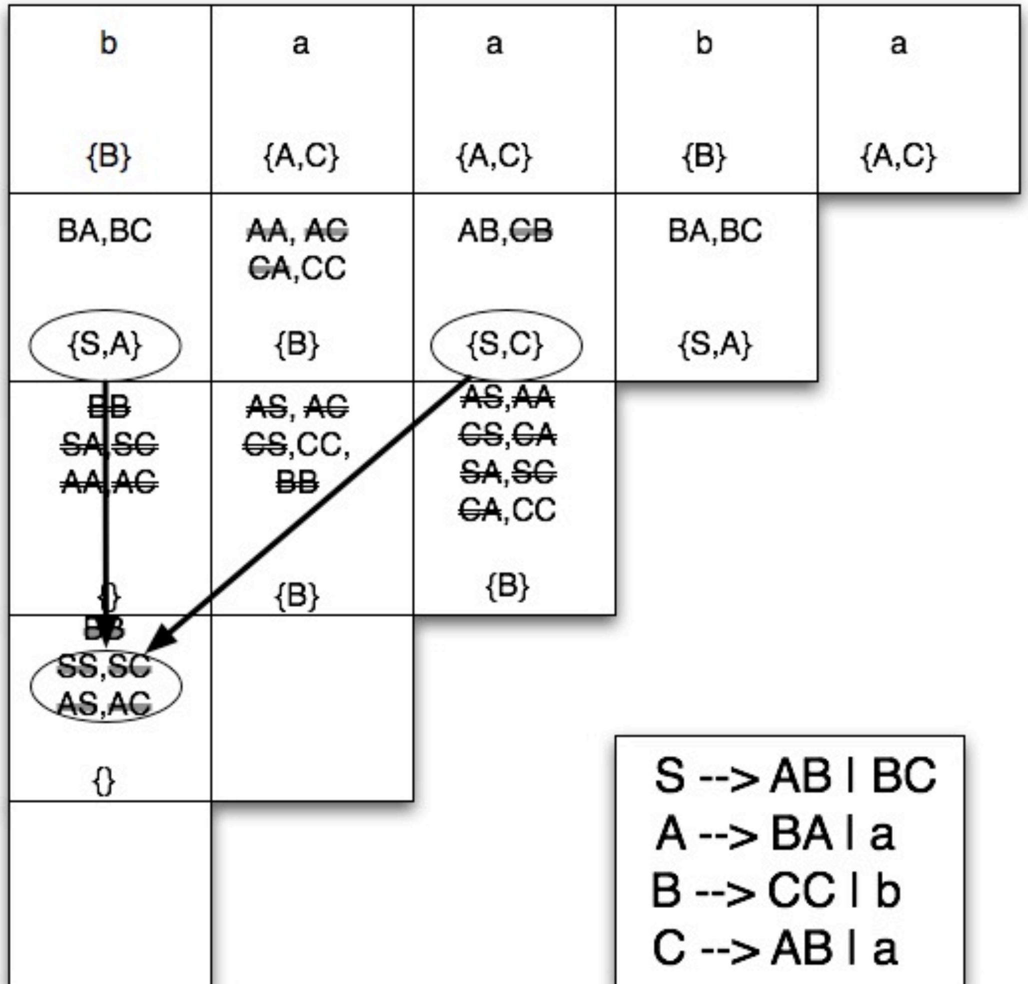


CYK Parser (Sequential Version)



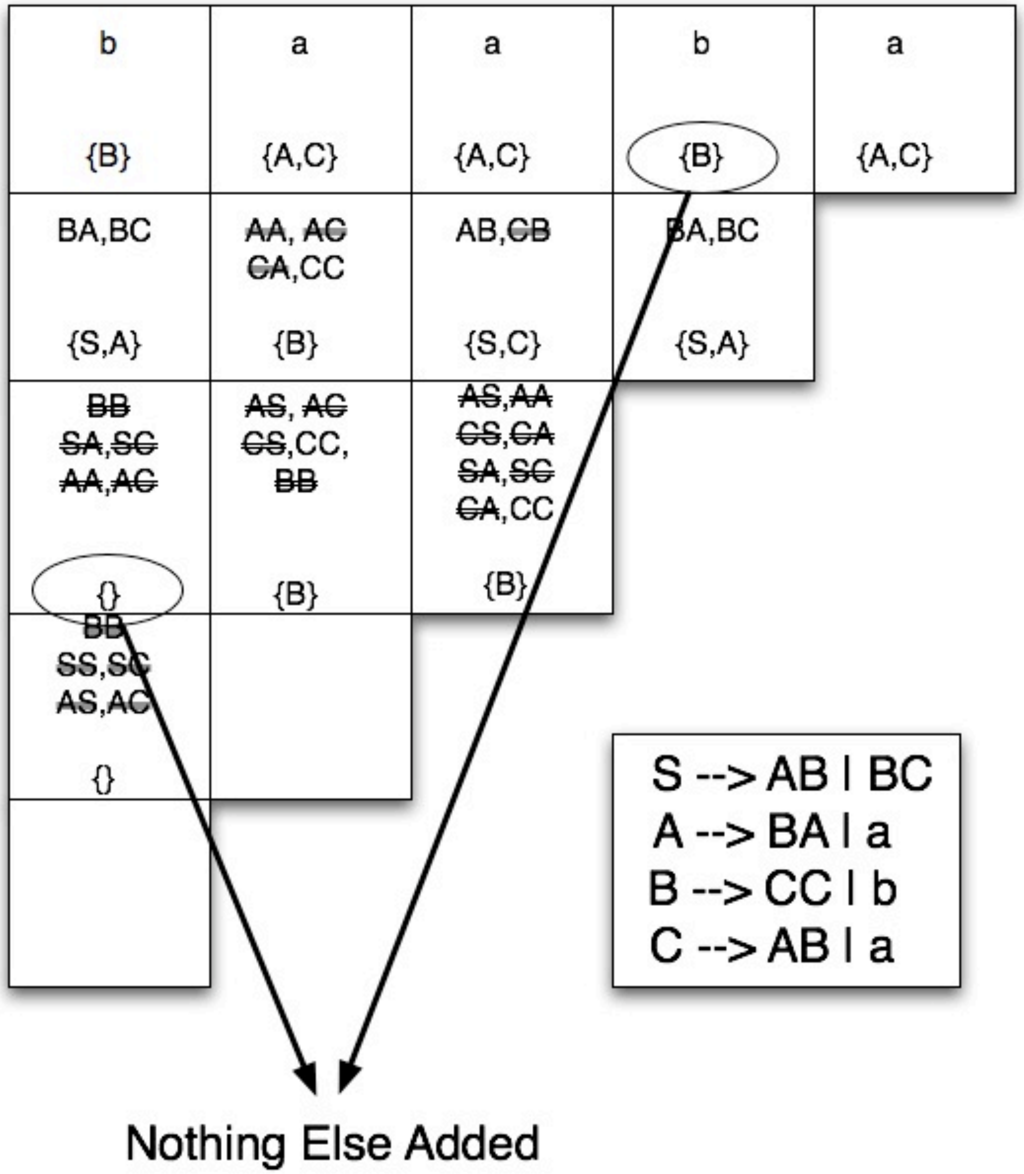
S --> AB | BC
A --> BA | a
B --> CC | b
C --> AB | a

CYK Parser (Sequential Version)

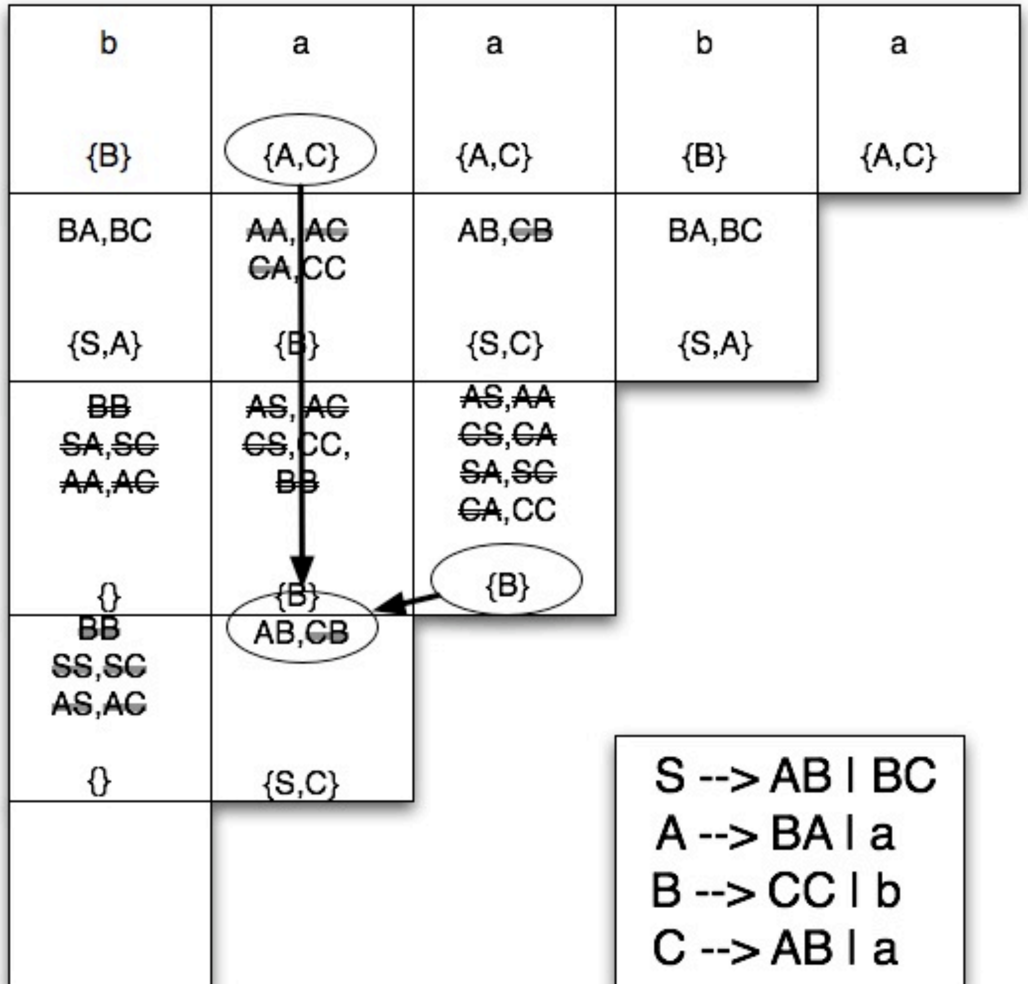


S --> AB | BC
A --> BA | a
B --> CC | b
C --> AB | a

CYK Parser (Sequential Version)

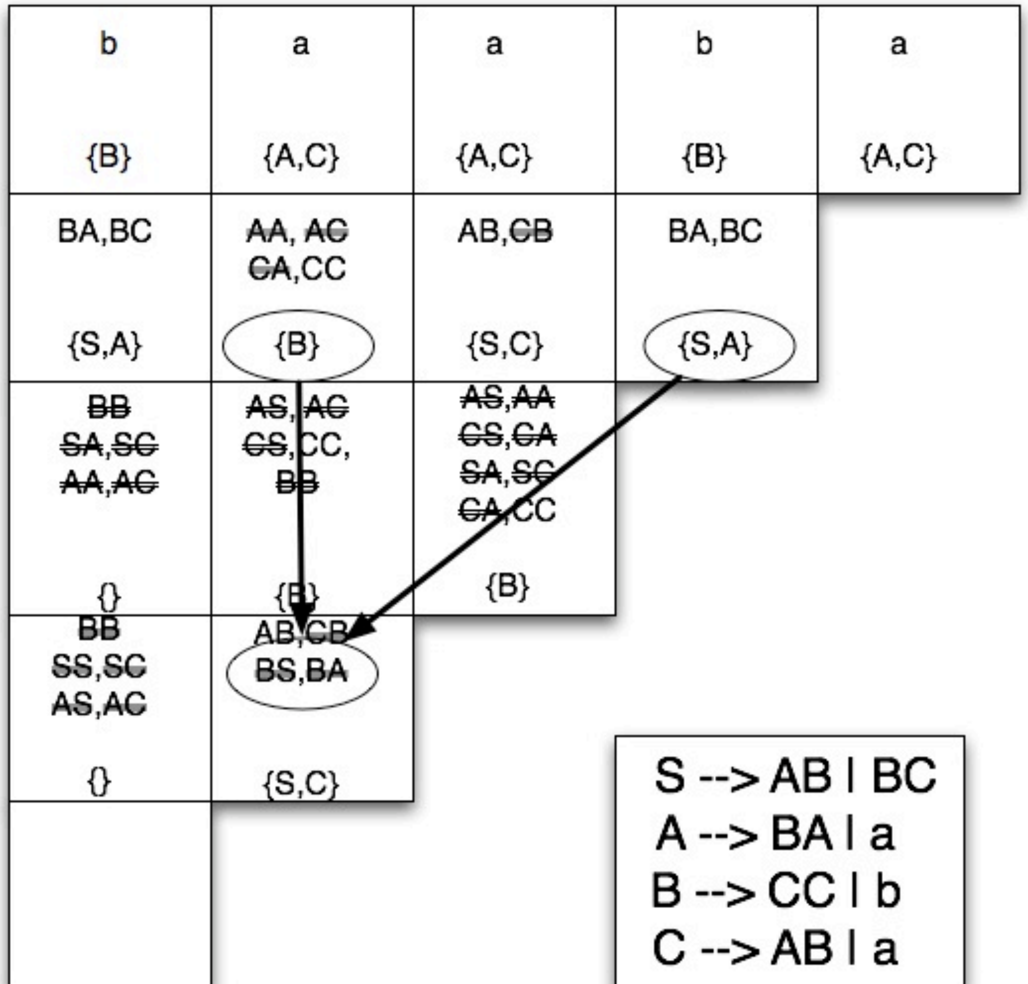


CYK Parser (Sequential Version)



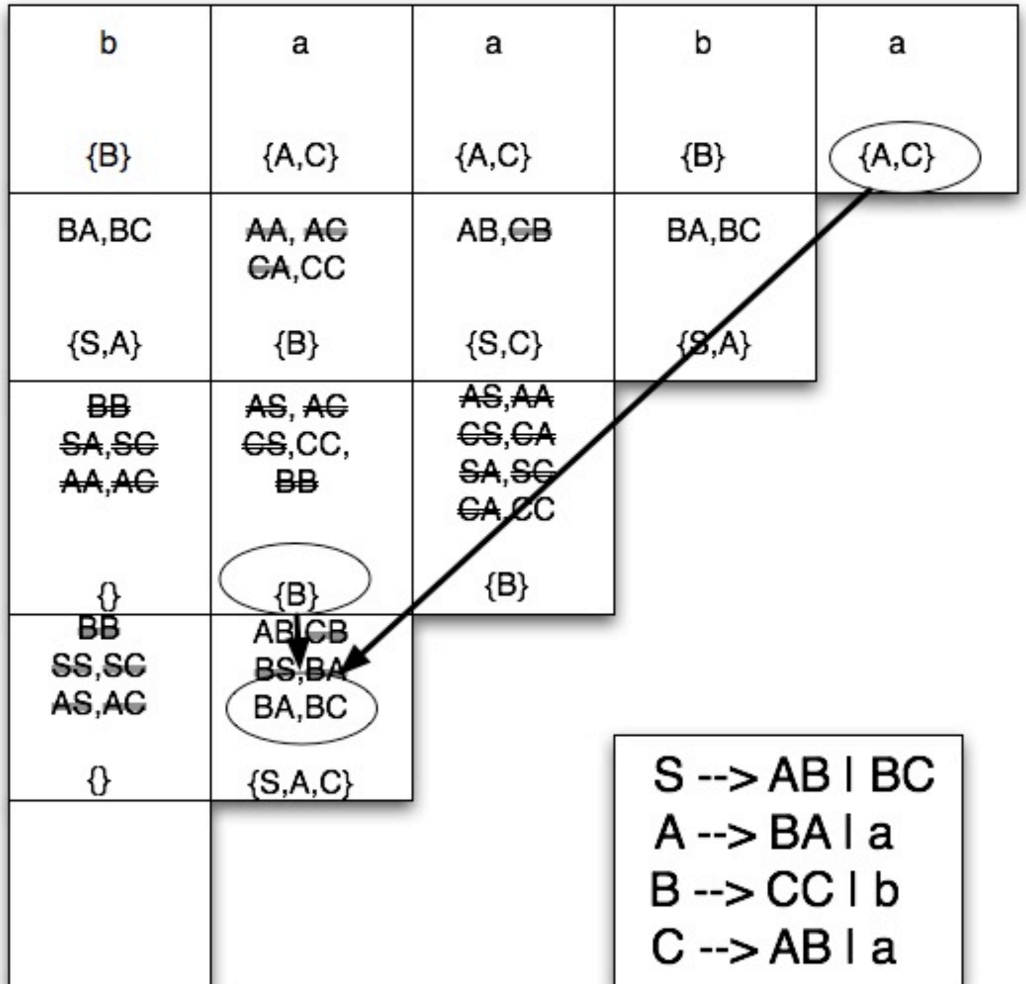
S --> AB | BC
A --> BA | a
B --> CC | b
C --> AB | a

CYK Parser (Sequential Version)



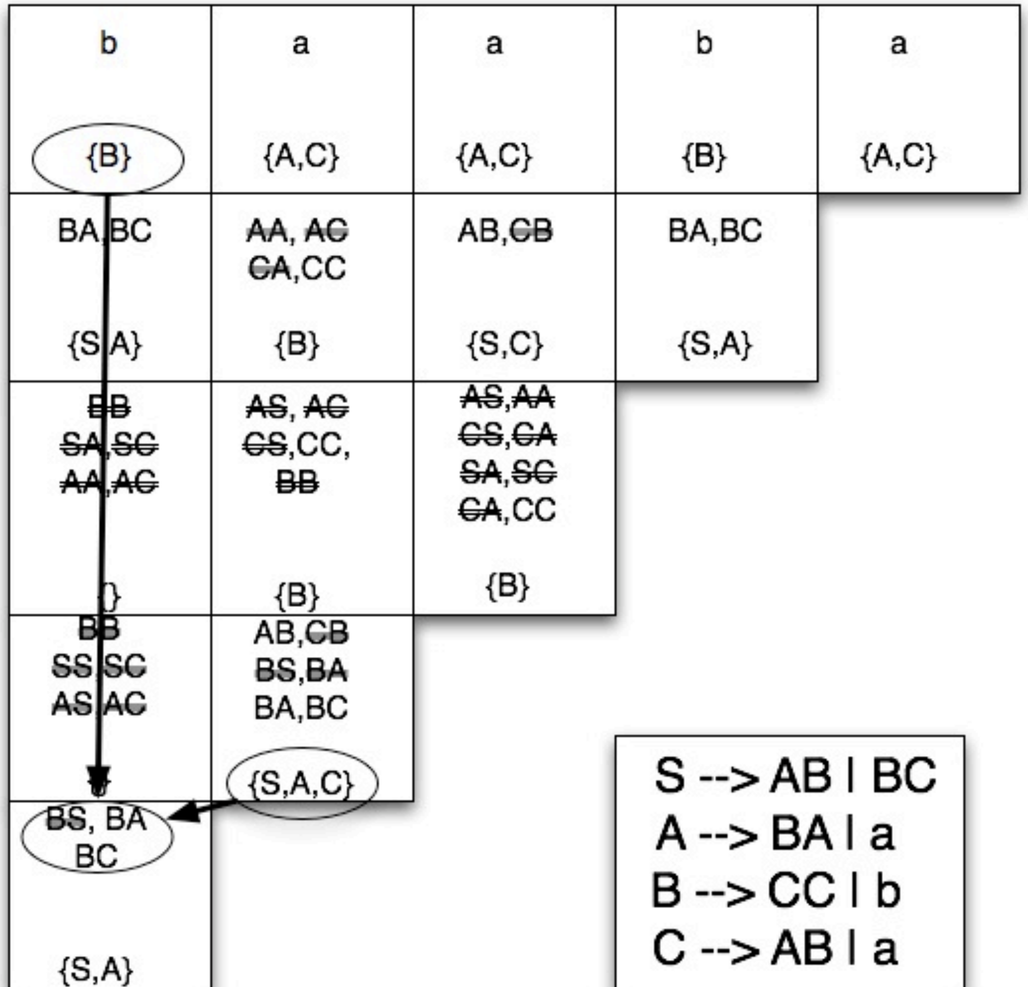
S --> AB | BC
A --> BA | a
B --> CC | b
C --> AB | a

CYK Parser (Sequential Version)

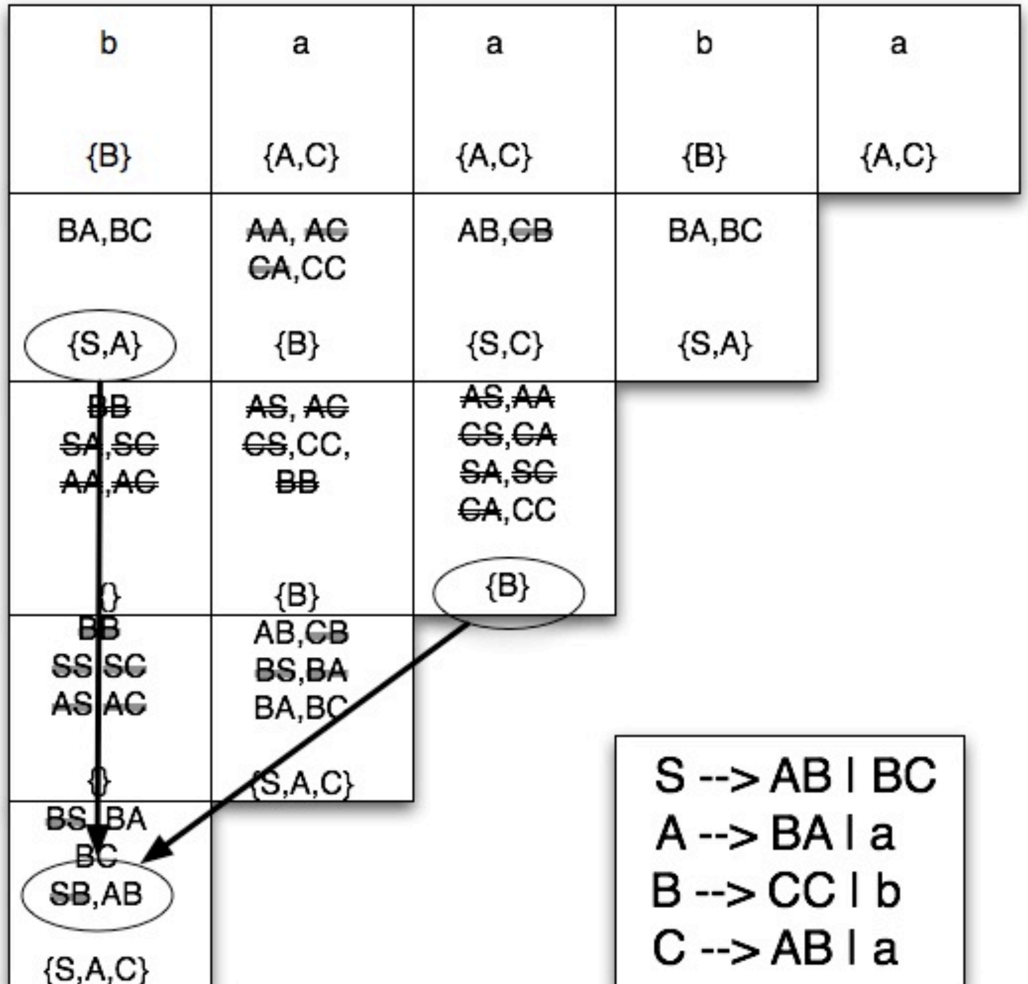


S --> AB | BC
A --> BA | a
B --> CC | b
C --> AB | a

CYK Parser (Sequential Version)

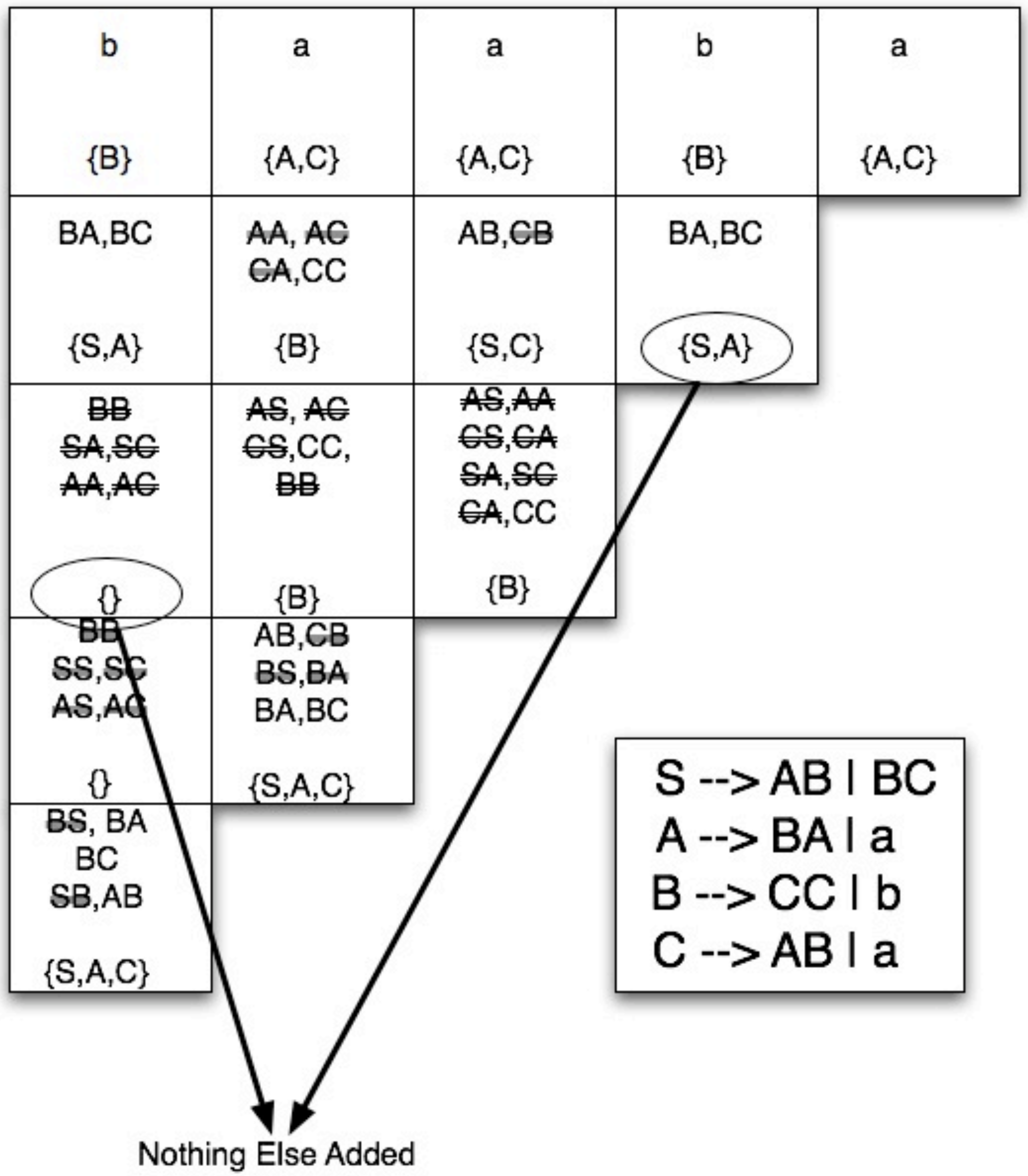


CYK Parser (Sequential Version)



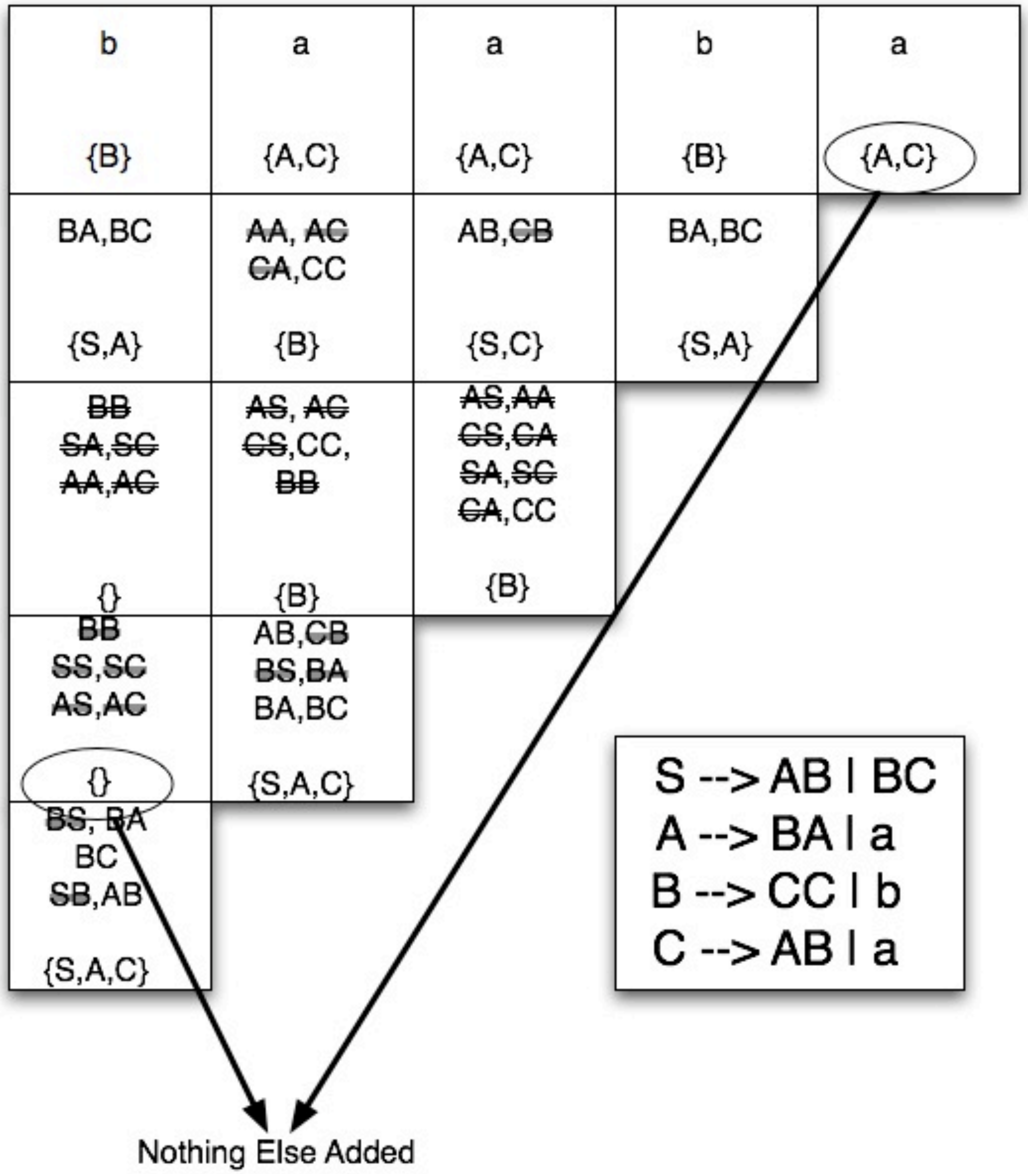
S --> AB | BC
A --> BA | a
B --> CC | b
C --> AB | a

CYK Parser (Sequential Version)



S --> AB | BC
A --> BA | a
B --> CC | b
C --> AB | a

CYK Parser (Sequential Version)



S --> AB | BC
A --> BA | a
B --> CC | b
C --> AB | a

CYK Parser (Sequential Version)

b {B}	a {A,C}	a {A,C}	b {B}	a {A,C}
BA,BC {S,A}	~~AA~~, ~~AC~~ ~~CA~~,CC {B}	AB,~~CB~~ {S,C}	BA,BC {S,A}	
~~BB~~ ~~SA~~,~~SC~~ ~~AA~~,~~AC~~ {}	~~AS~~, ~~AC~~ ~~CS~~,CC, ~~BB~~ {B}	~~AS~~,~~AA~~ ~~CS~~,~~CA~~ ~~SA~~,~~SC~~ ~~CA~~,CC {B}		
~~BB~~ ~~SS~~,~~SC~~ ~~AS~~,~~AC~~ {}	AB,~~CB~~ ~~BS~~,~~BA~~ BA,BC {S,A,C}			
~~BS~~, BA BC ~~SB~~,AB {S,A,C}				

S --> AB | BC
A --> BA | a
B --> CC | b
C --> AB | a

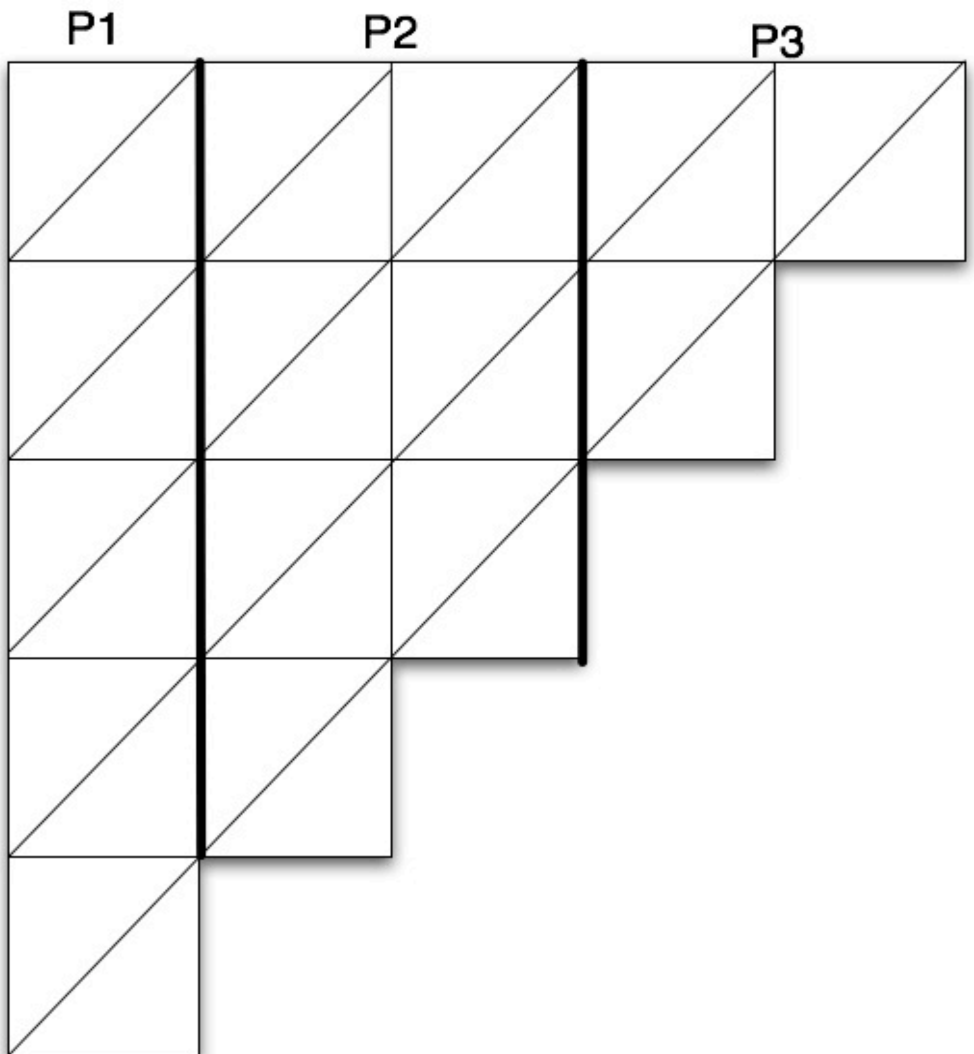
The CYK Parser Algorithm (Sequential Version)

```
(* for the first row *)
  1) for i := 1 to n do
  2)     V_i1 := { A | A --> a is a production
                   rule and the ith symbol of s
                   is a}

(* for subsequent rows *)
  3) for j := 2 to n do
  4)     for i := 1 to (n - j + 1) do
  5)         V_ij := {}
  6)         for k := 1 to (j - 1) do
  7)             V_ij := V_ij U { A | A --> BC
                                  is a production
                                            rule,
                                  B is in V_ik,
                                  C is in
                                    V_i+k,j-k }
```

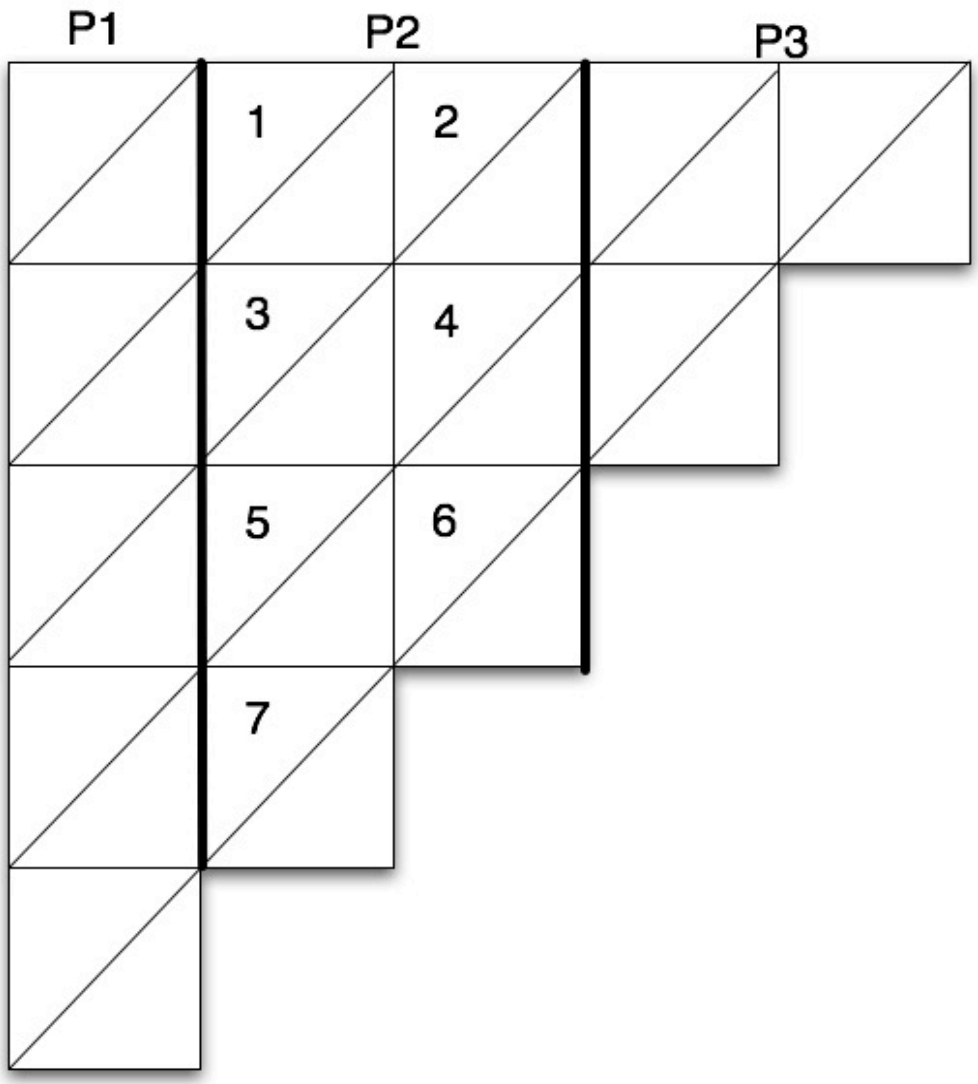
Figure3: Pseudo-code for the sequential CYK algorithm. Adapted from Hopcroft, Ullman, 1979, pp139-140.

CYK Parser (Parallel Version)



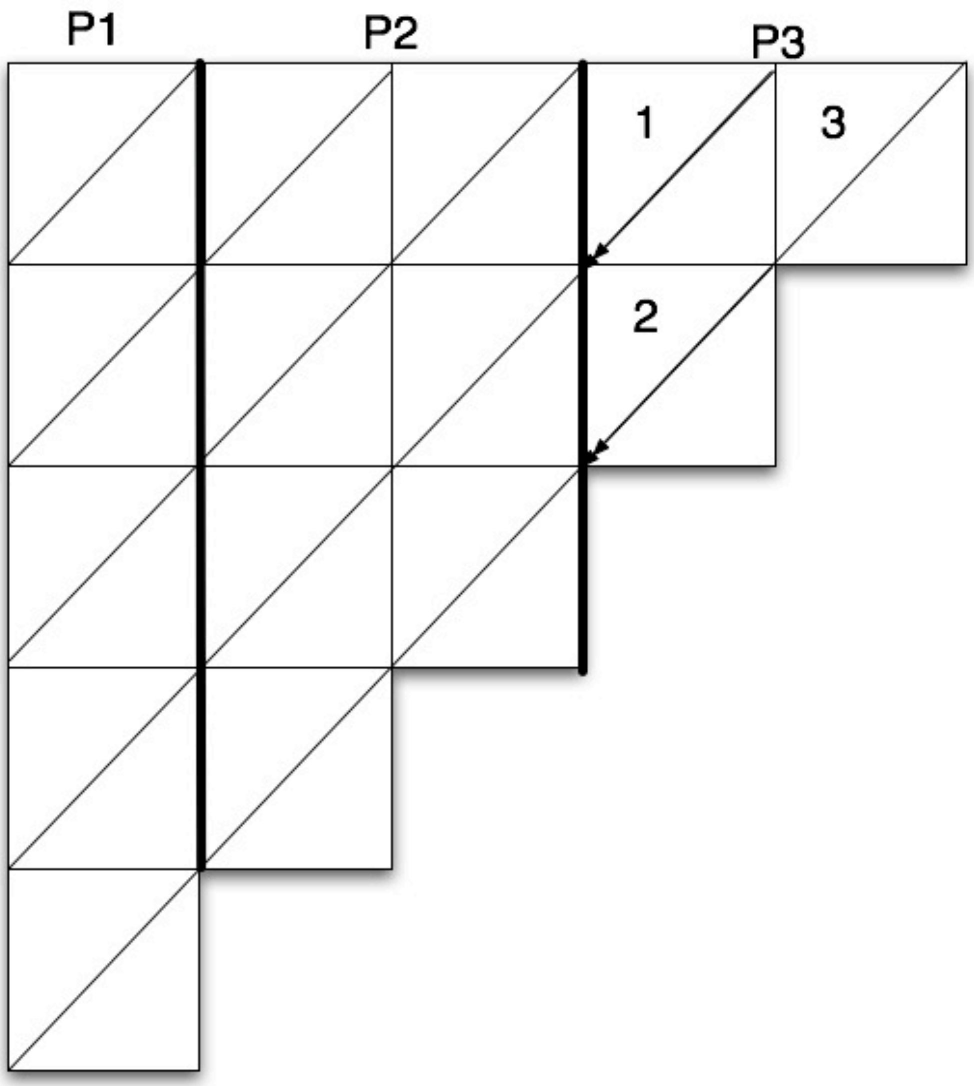
Matrix for a string of length 5 using 3 processors

CYK Parser (Parallel Version)



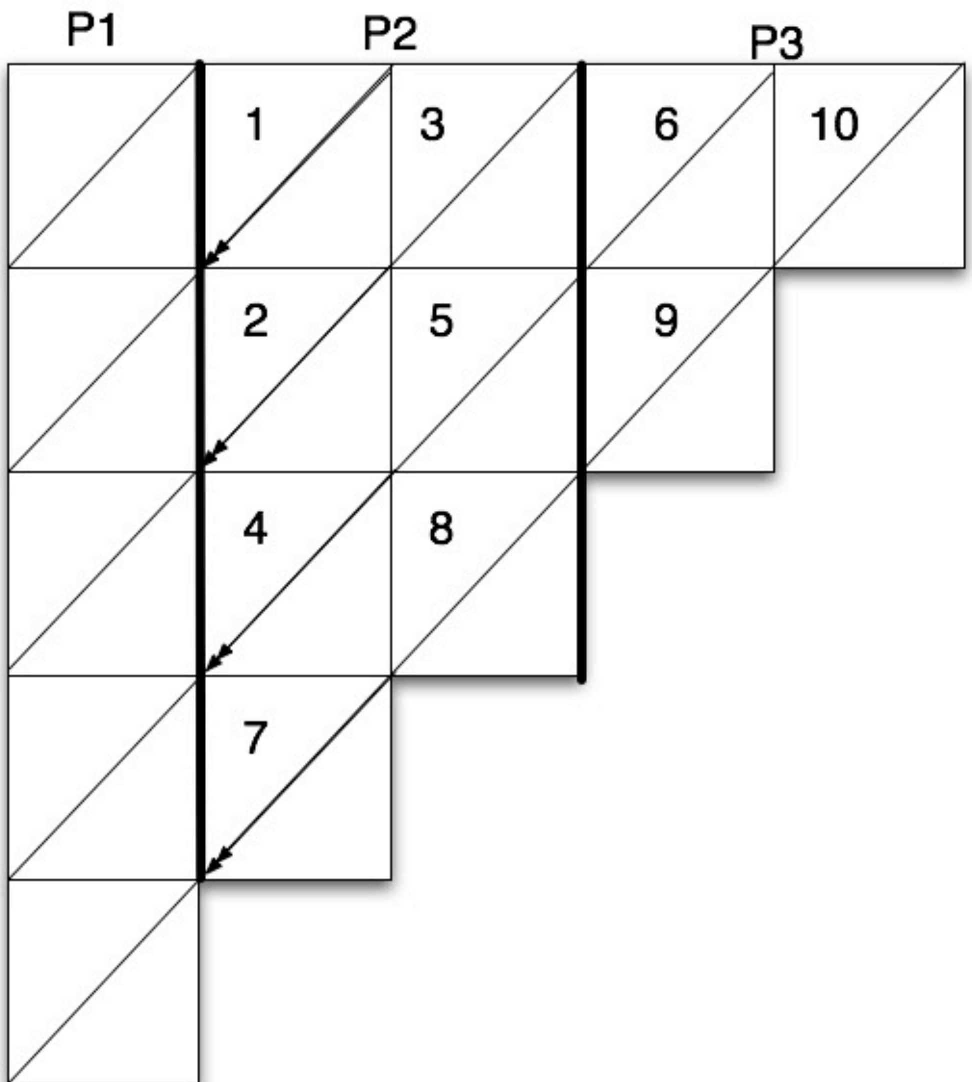
Order of calculation for processor P2. P2 calculates a diagonal at a time.

CYK Parser (Parallel Version)



Order of information received by P2. P2 receives a diagonal at a time.

CYK Parser (Parallel Version)



Order of information P2 sends to P1. P2 sends a diagonal at a time.

The CYK Parser Algorithm (Parallel Version)

if not last processor send all along to P_{i+1}

let $l = \sum_{q=1}^{p}$ length of substring for P_q

for j := 1 to l do

  if necessary get diagonal from p_{i+1}

  let m = length of the diagonal within P_i

  for k := 1 to m do

    calculate $V_{j-k+1,k}$

  if i <> 1

    then send back new diagonal to P_{i-1}

    else send back $V_{1,n}$ to Host