Parsing VI
The Last Parsing Lecture

Quiz : The Left Recursive SheepNoise Grammar

1. Goal → SheepNoise
2. SheepNoise → SheepNoise baa
3. | baa

Example From SheepNoise

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

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

Item	*From*
[*Goal*→•*SheepNoise*,EOF]	Original item
[*SheepNoise*→•*SheepNoise* baa,EOF]	1, δa is EOF
[*SheepNoise*→ • baa,EOF]	1, δa is EOF
[*SheepNoise*→•*SheepNoise* baa,baa]	2, δa is baa EOF
[*SheepNoise*→ • baa,baa]	2, δa is baa EOF

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

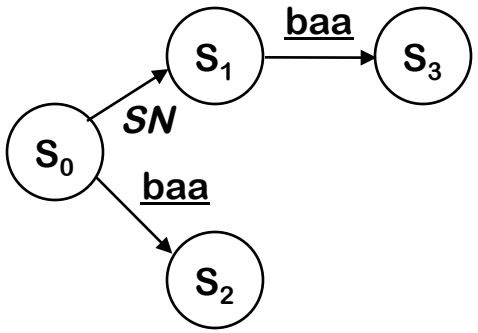
S_0 : { [*Goal*→ • *SheepNoise*, EOF], [*SheepNoise*→ • *SheepNoise* baa, EOF],
[*SheepNoise*→ • baa, EOF], [*SheepNoise*→ • *SheepNoise* baa, baa],
[*SheepNoise*→ • baa, baa] }

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] }

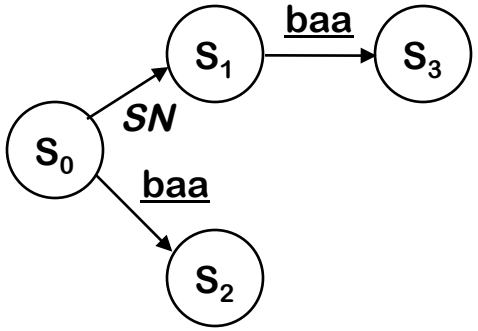
Control DFA for SN



Filling in the ACTION and GOTO Tables

∀ set $s_x \in S$
  ∀ 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 $[S' \rightarrow S \cdot, EOF]$
      then ACTION$[x, \underline{a}] \leftarrow$ "*accept*"
    else if i is $[A \rightarrow \beta \cdot, \underline{a}]$
      then ACTION$[x, \underline{a}] \leftarrow$ "*reduce* $A \rightarrow \beta$"
  ∀ $n \in NT$
    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] }

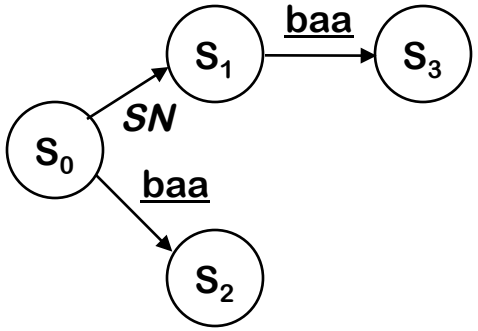
S_1 = *Goto*(S_0 , ***SheepNoise***) =
{ [*Goal*→ *SheepNoise* •, EOF], [*SheepNoise*→ *SheepNoise* • baa, EOF],
[*SheepNoise*→ *SheepNoise* • baa, baa] }

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

Filling in the ACTION and GOTO Tables

∀ *set* $s_x \in S$
  ∀ *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* $[S' \rightarrow S \cdot, EOF]$
      then ACTION$[x, \underline{a}] \leftarrow$ "*accept*"
    else if *i is* $[A \rightarrow \beta \cdot, \underline{a}]$
      then ACTION$[x, \underline{a}] \leftarrow$ "*reduce* $A \rightarrow \beta$"
  ∀ $n \in NT$
    if $goto(s_x, n) = s_k$
      then GOTO$[x, n] \leftarrow k$

Control DFA for SN



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

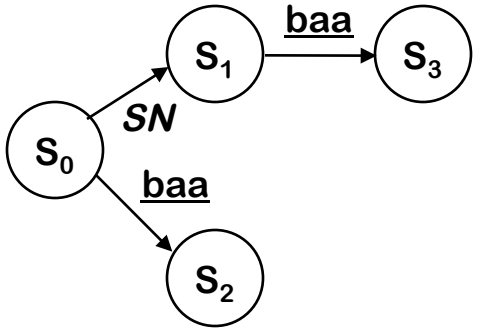
S_2 = *Goto*(S_0 , **baa**) = { [***SheepNoise*** → **baa** ·, **EOF**],
[***SheepNoise*** → **baa** ·, **baa**] }

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

Filling in the ACTION and GOTO Tables

∀ *set* $s_x \in S$
  ∀ *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* $[S' \rightarrow S \cdot, EOF]$
      then ACTION$[x, \underline{a}] \leftarrow$ "*accept*"
    else if *i is* $[A \rightarrow \beta \cdot, \underline{a}]$
      then ACTION$[x, \underline{a}] \leftarrow$ "*reduce* $A \rightarrow \beta$"
  ∀ $n \in NT$
    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*→β•a γ,b] and [*B*→β•,a] ?

- First item generates "shift", second generates "reduce"
- Both define ACTION[s,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 worke example

What if set *s* contains [*A*→γ•, a] and [*B*→γ•, a] ?

- Each generates "reduce", but with a different production
- Both define ACTION[s,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

	Advantages	*Disadvantages*
Top-down recursive descent	Fast Good locality Simplicity Good error detection	Hand-coded High maintenance Right associativity
LR(1)	Fast (Direct encoding) Deterministic langs. Automatable Left associativity	Large working sets Poor error messages Large table sizes

CYK Parser

- Simple context-free-language parser
  - running time is $O(n^3)$, space is $O(n^2)$

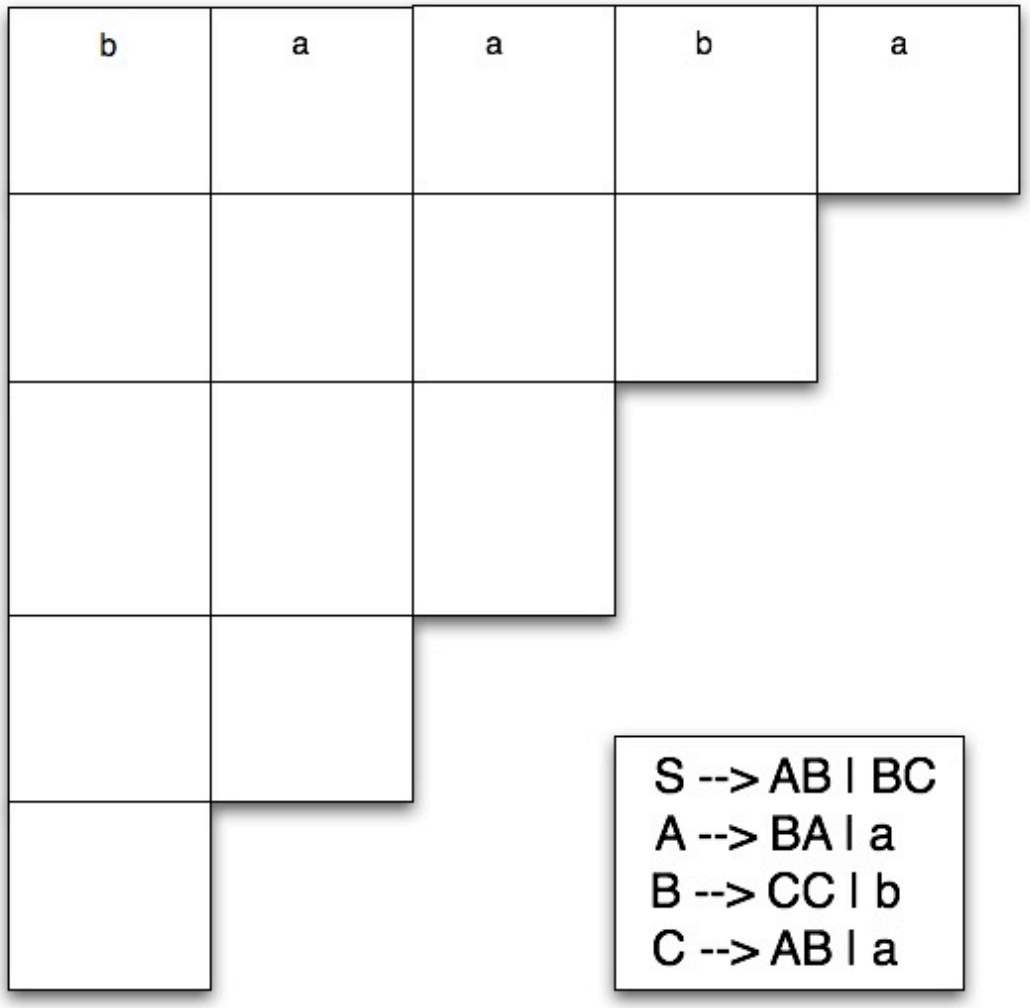
- Shunned for many years

  "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

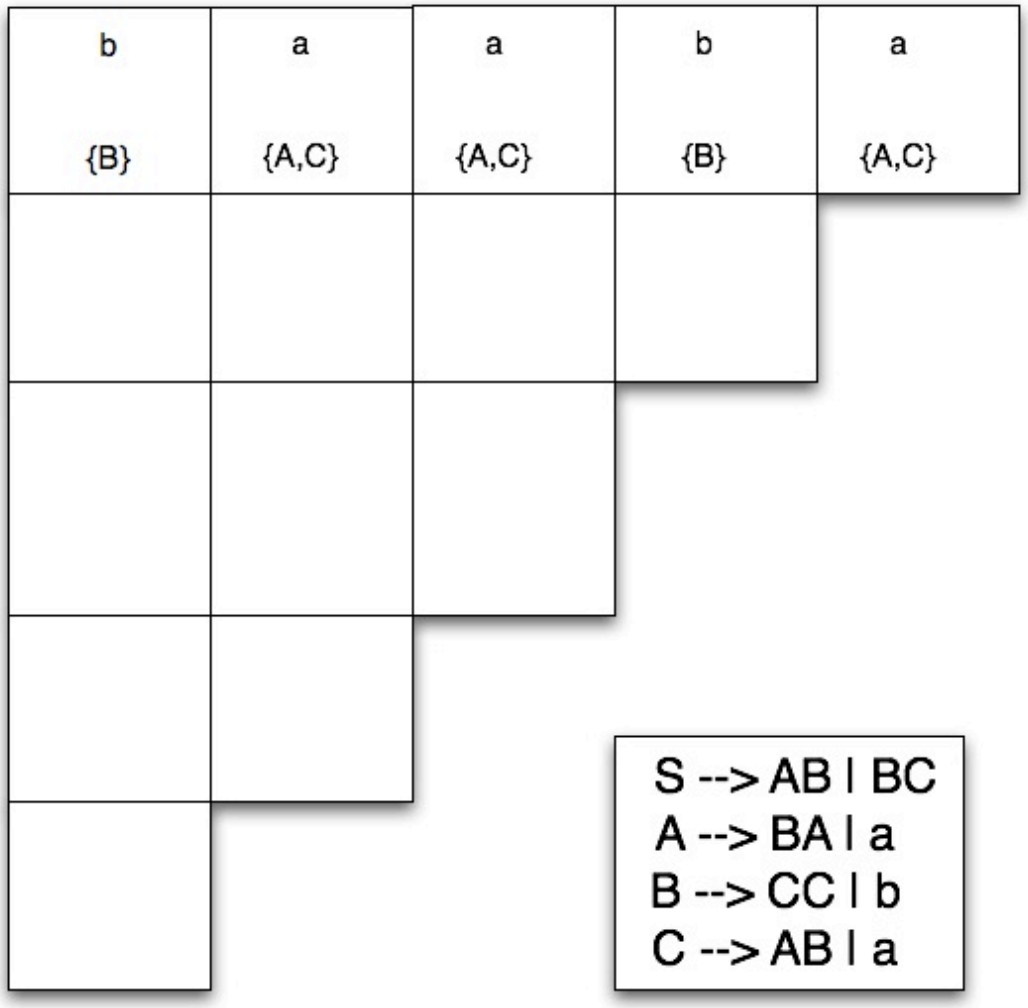
- But in practice, running time is more like $O(n^{\approx 1.2})$
  - Plus computers are now 1,000,000-times faster than in 1972

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

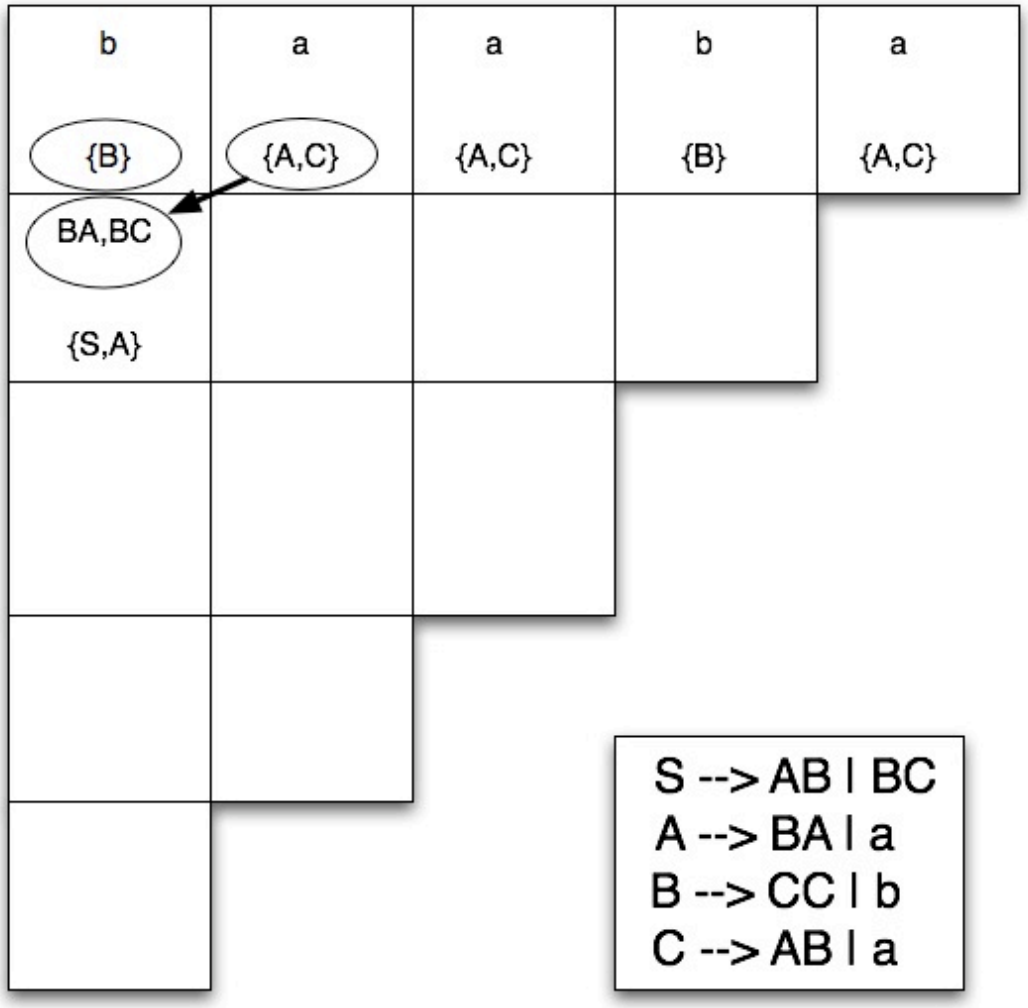
CYK Parser (Sequential Version)



CYK Parser (Sequential Version)

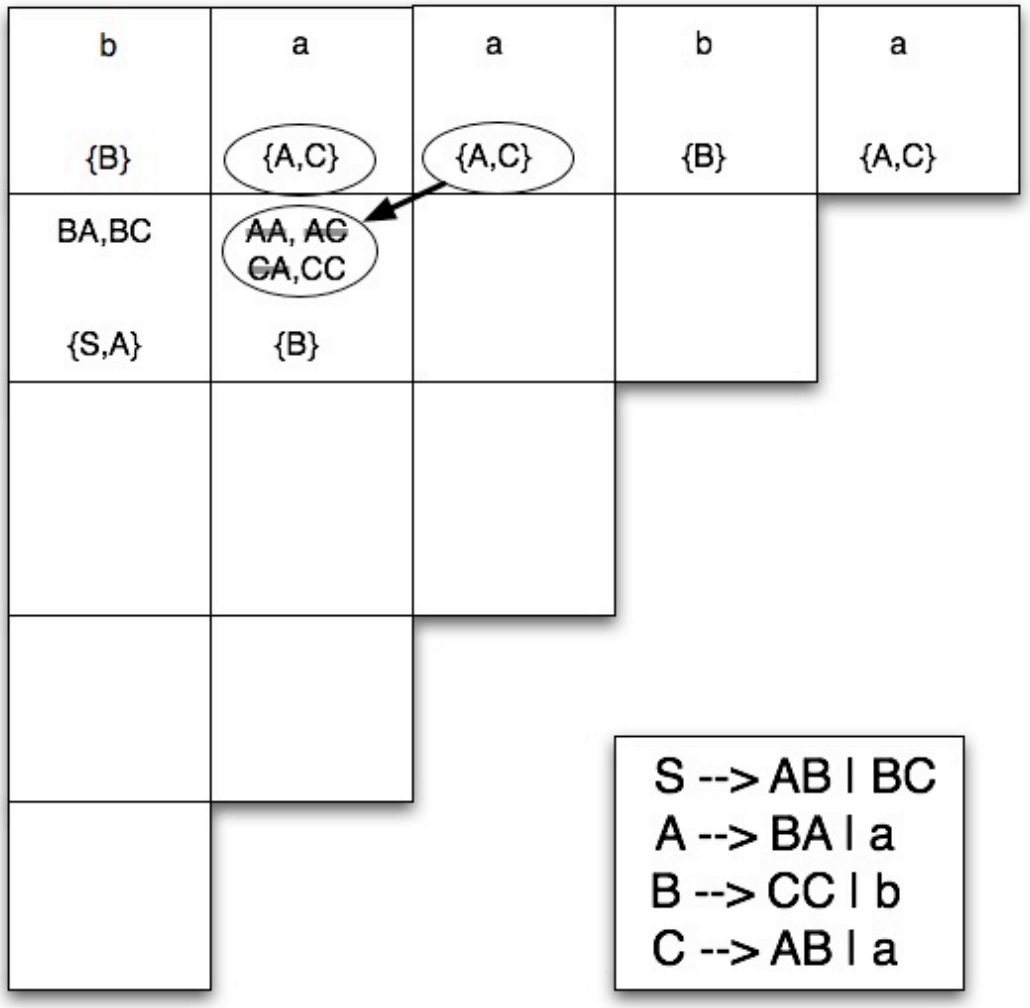


CYK Parser (Sequential Version)

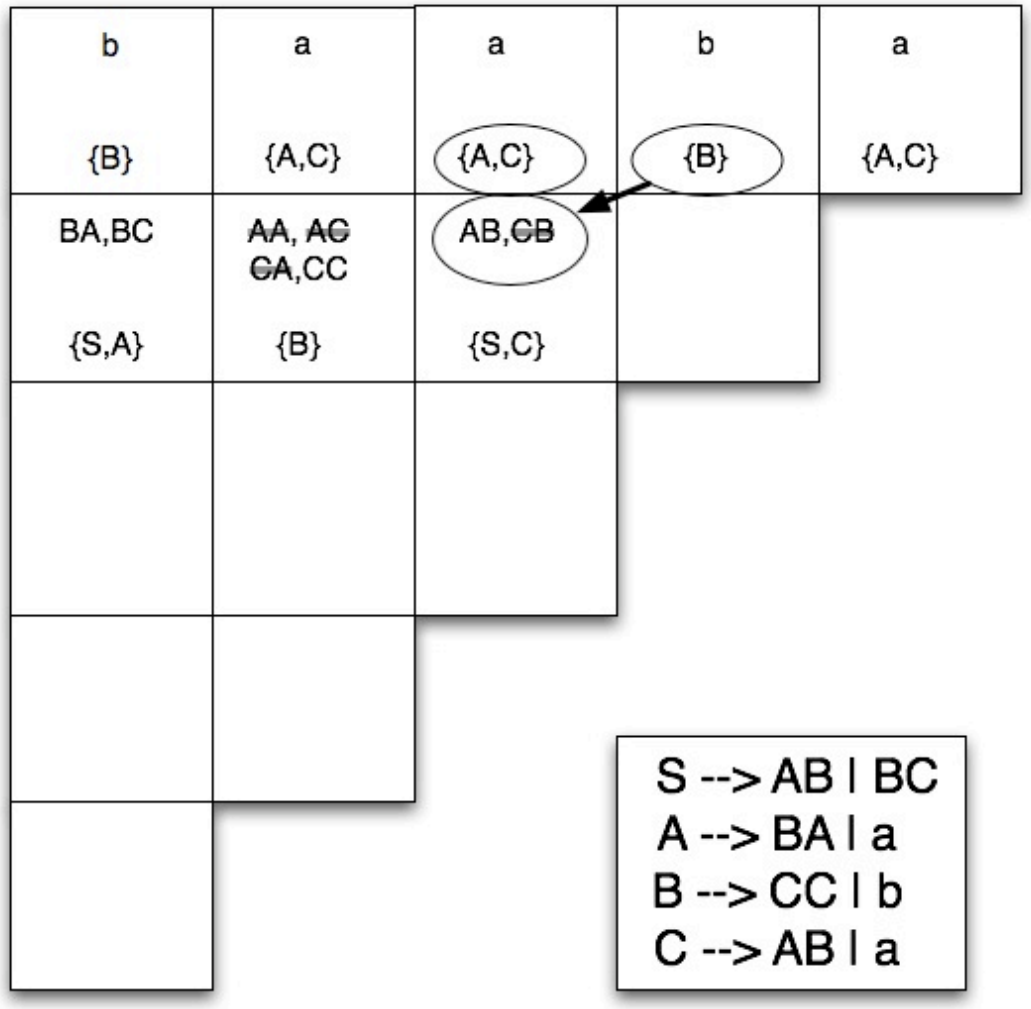


S --> AB | BC
A --> BA | a
B --> CC | b
C --> AB | a

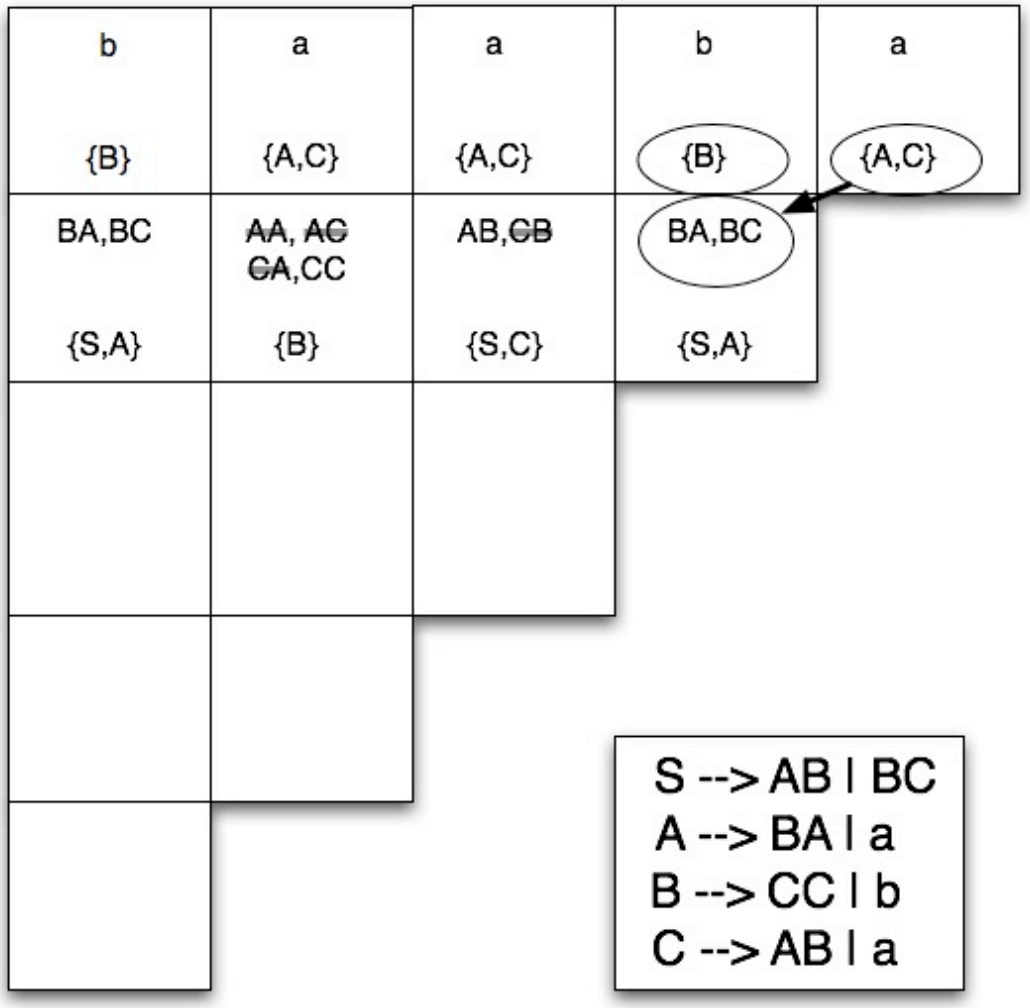
CYK Parser (Sequential Version)



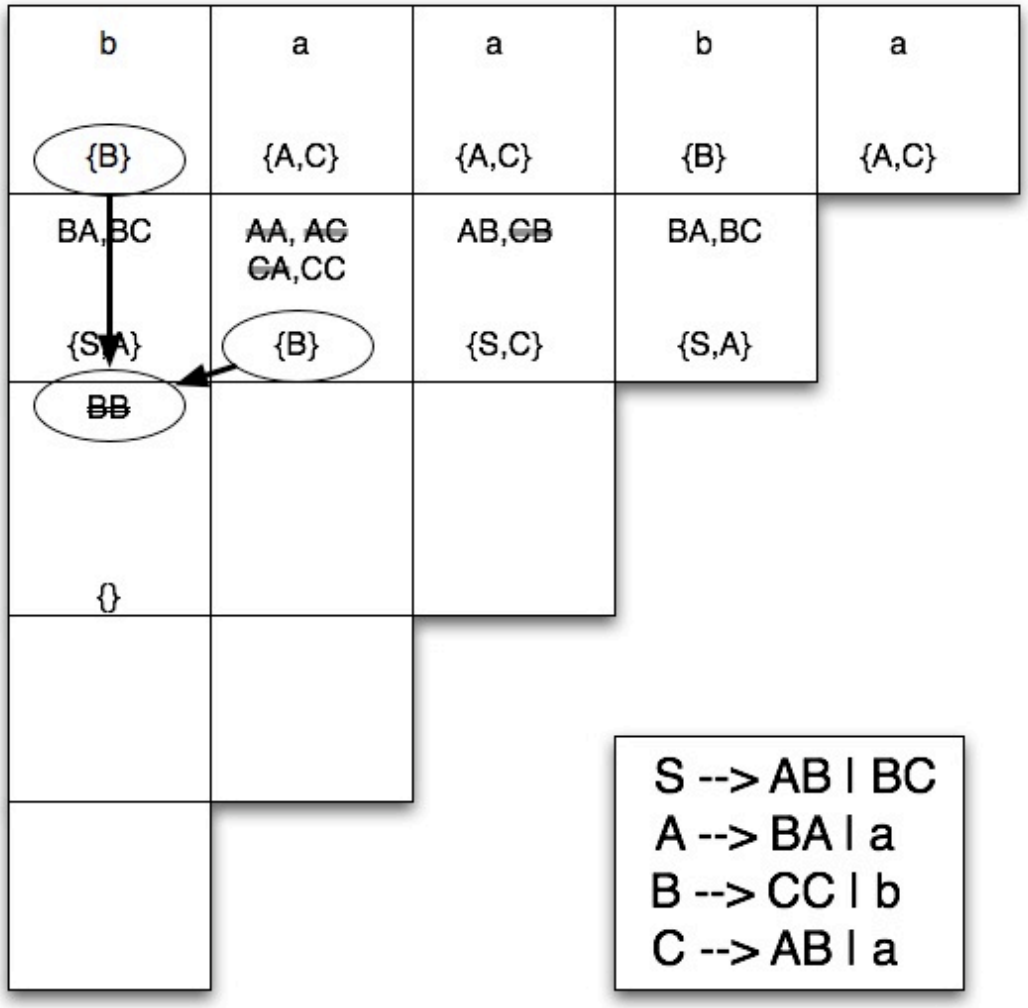
CYK Parser (Sequential Version)



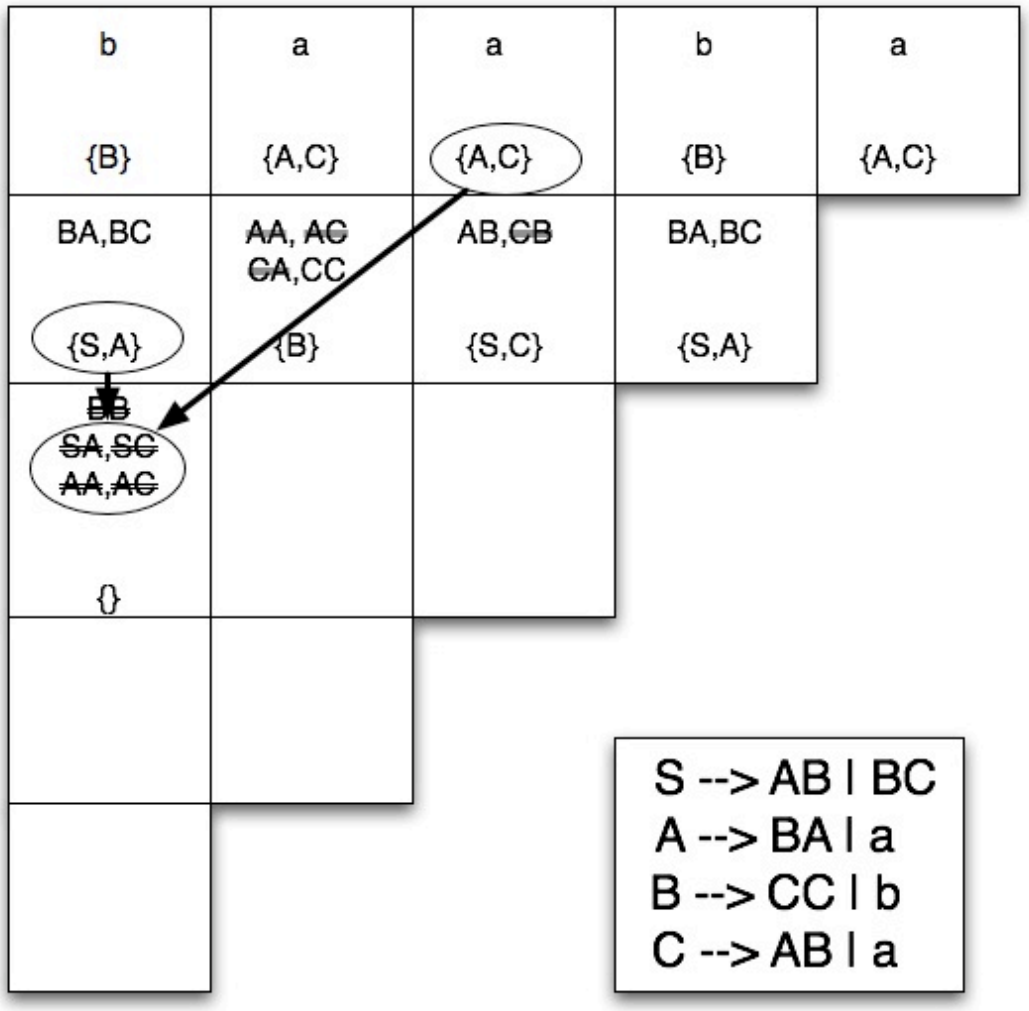
CYK Parser (Sequential Version)



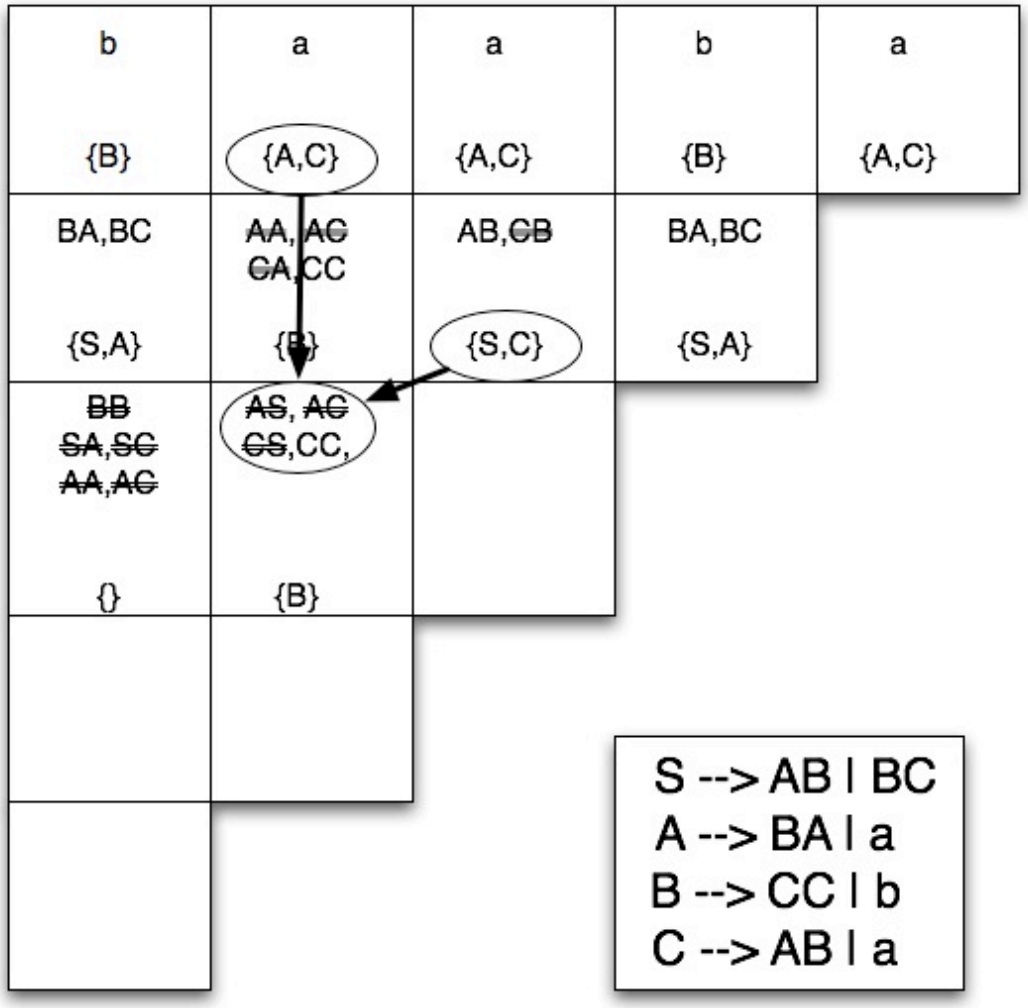
CYK Parser (Sequential Version)



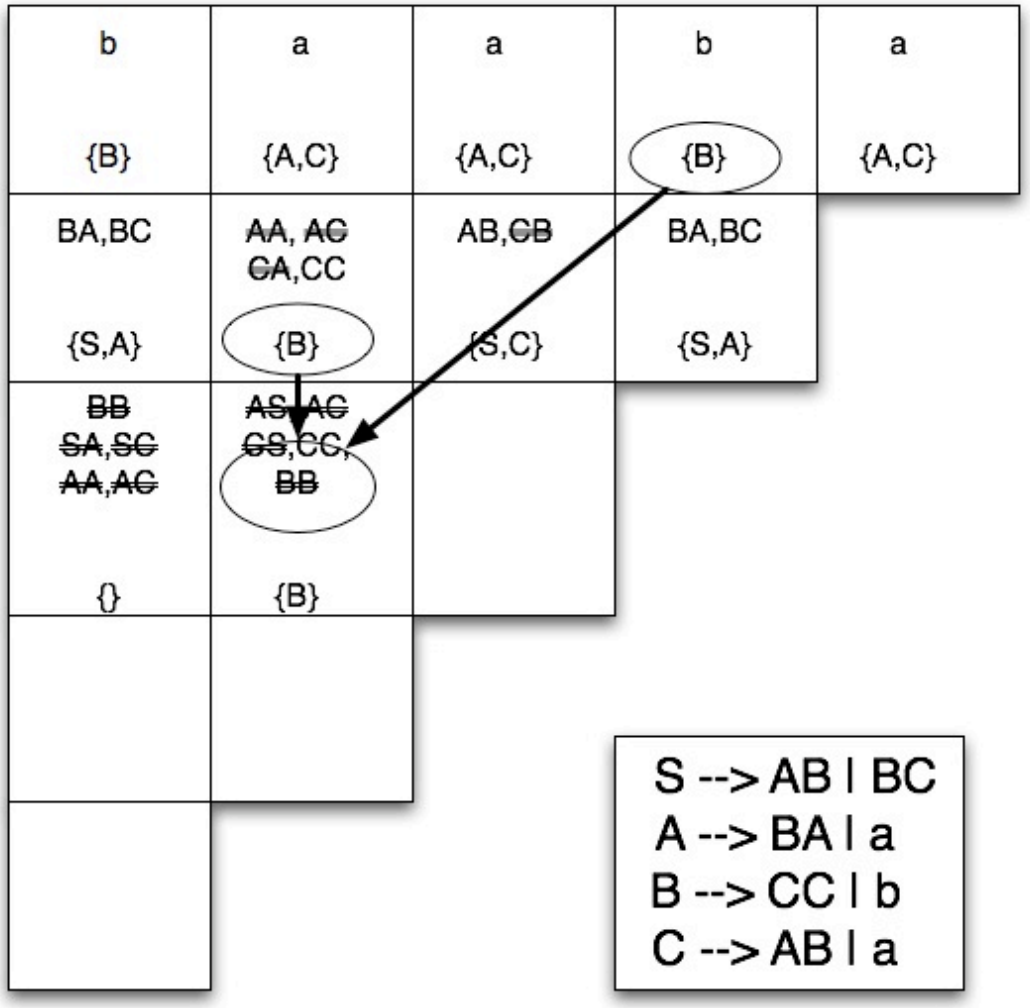
CYK Parser (Sequential Version)



CYK Parser (Sequential Version)

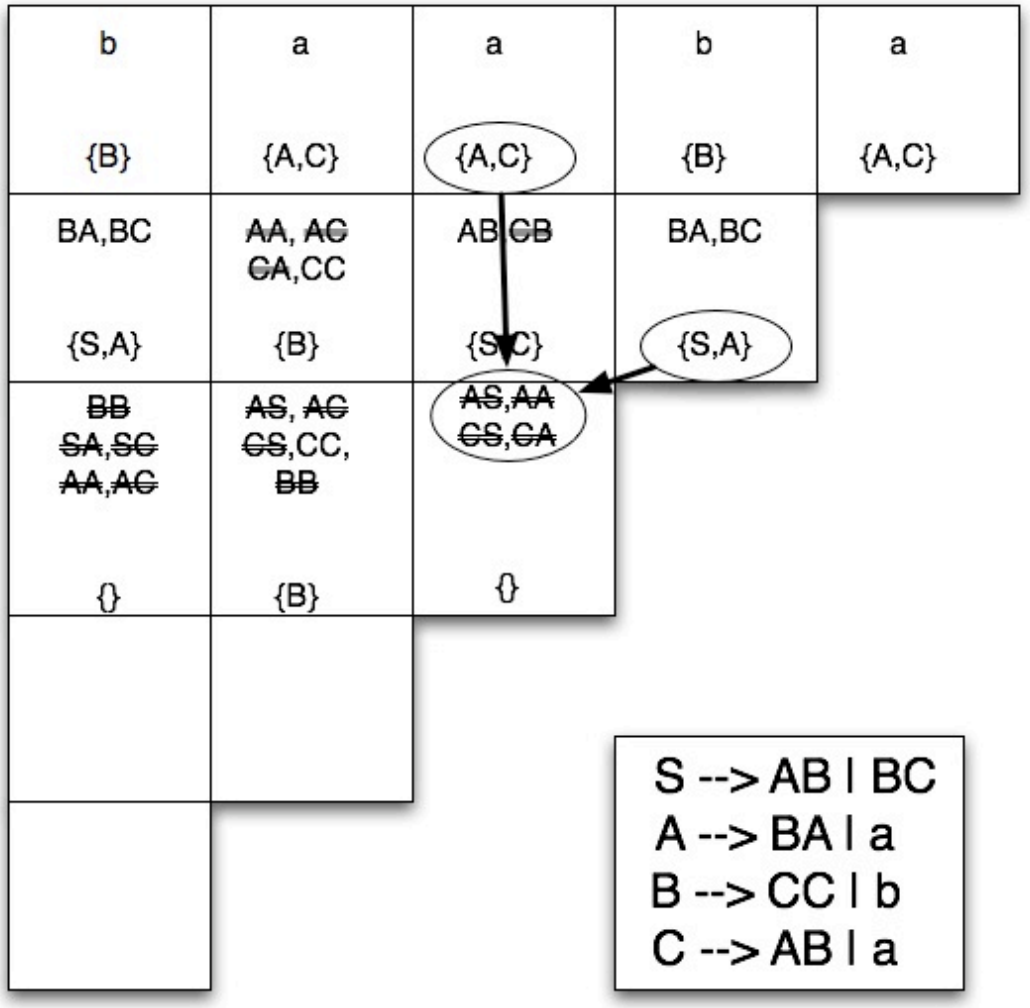


CYK Parser (Sequential Version)

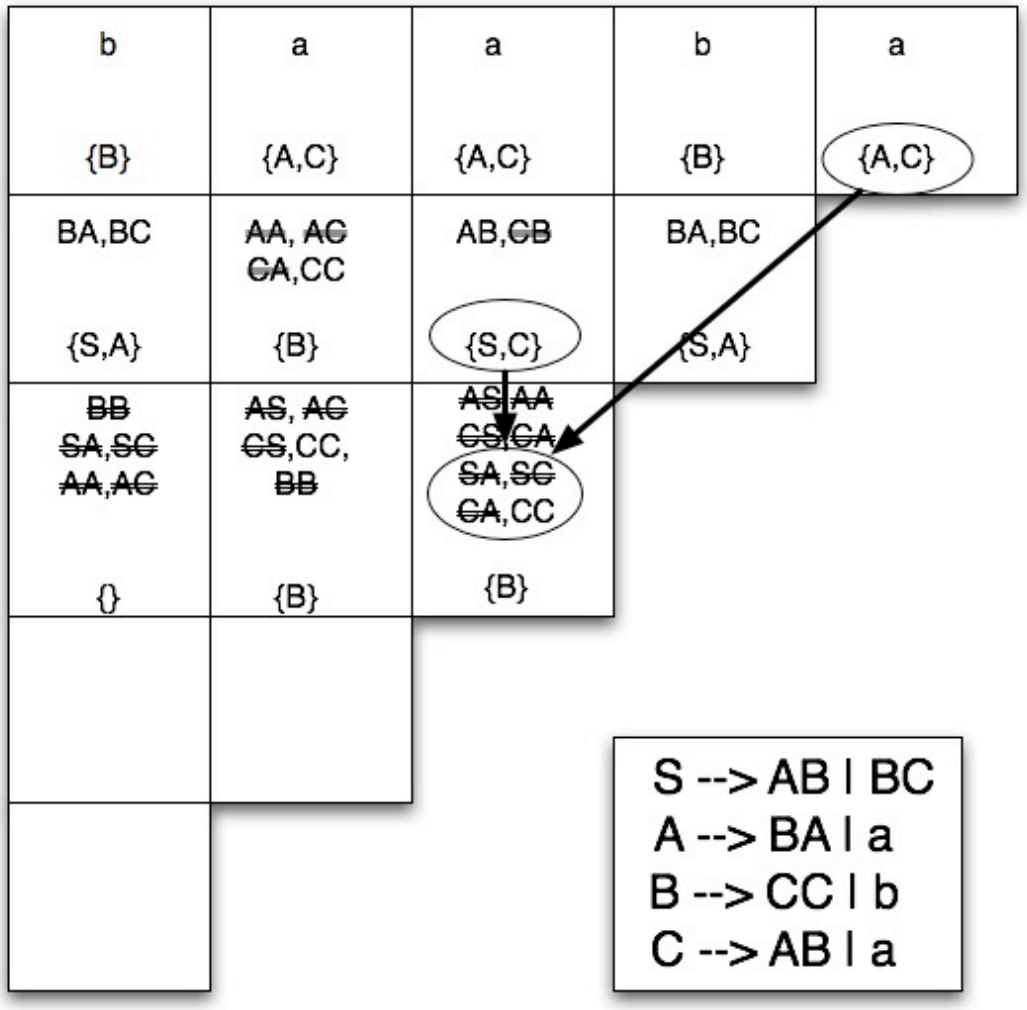


S --> AB | BC
A --> BA | a
B --> CC | b
C --> AB | a

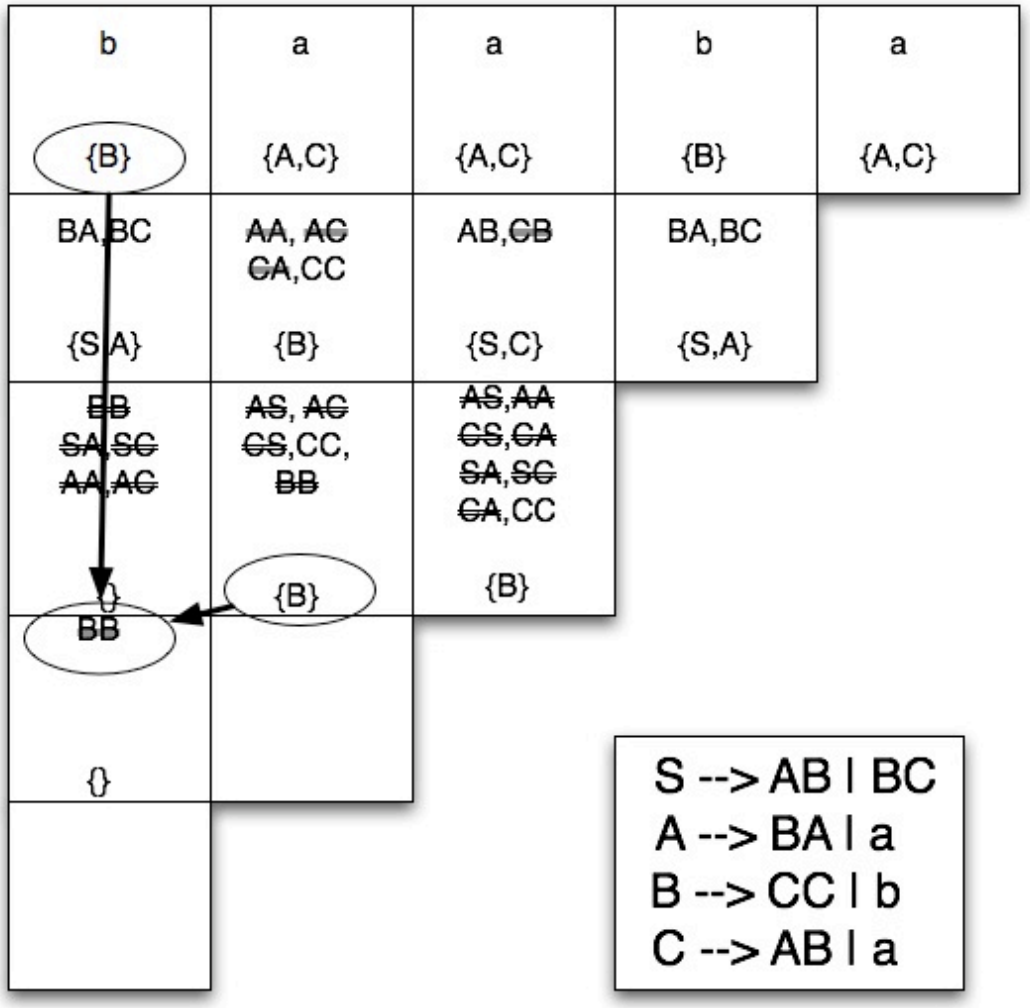
CYK Parser (Sequential Version)



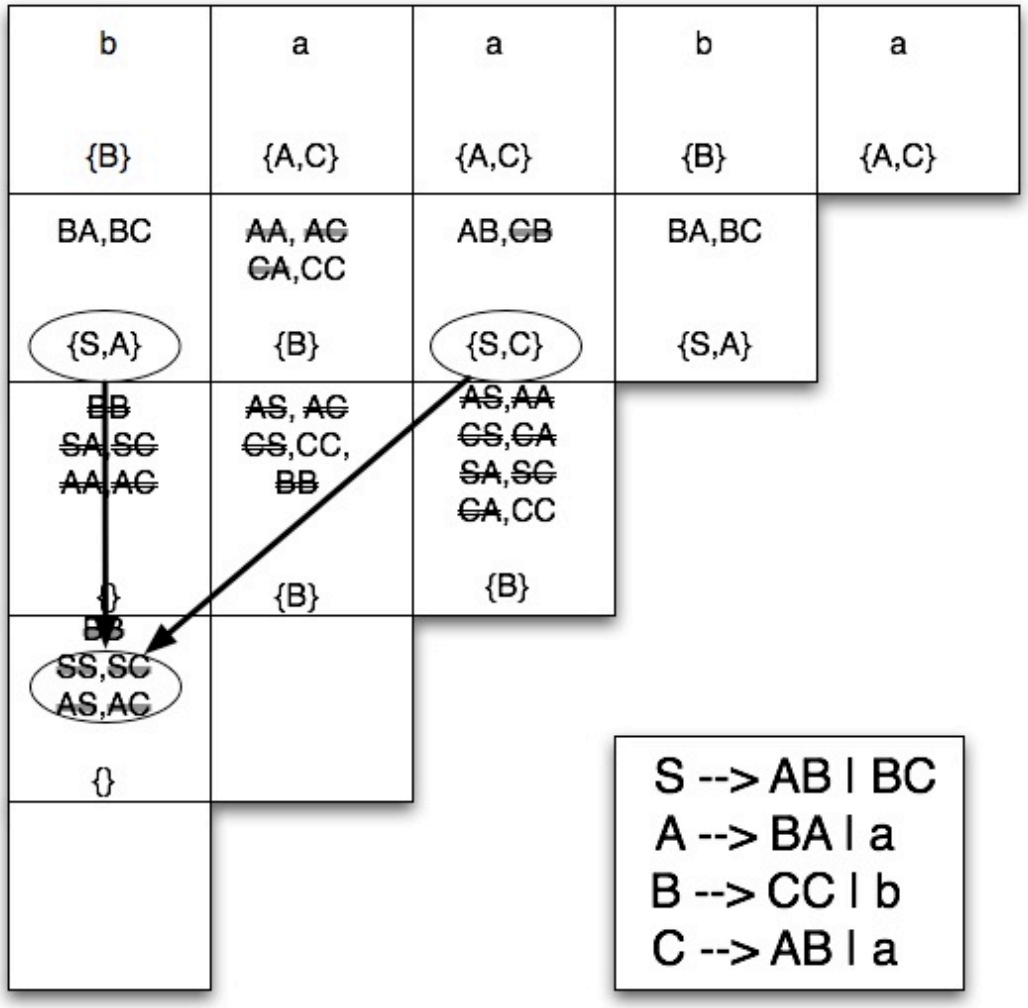
CYK Parser (Sequential Version)



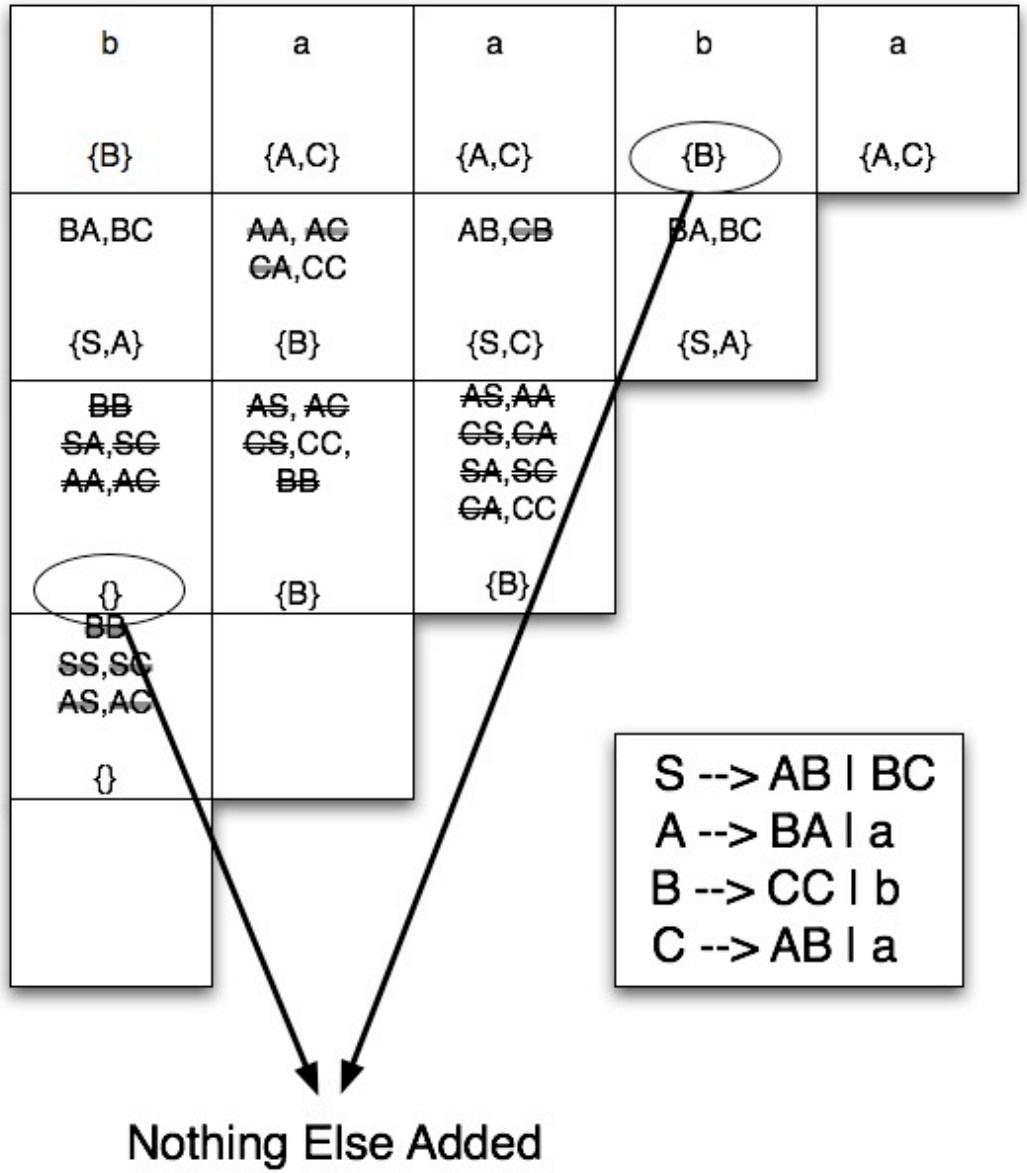
CYK Parser (Sequential Version)



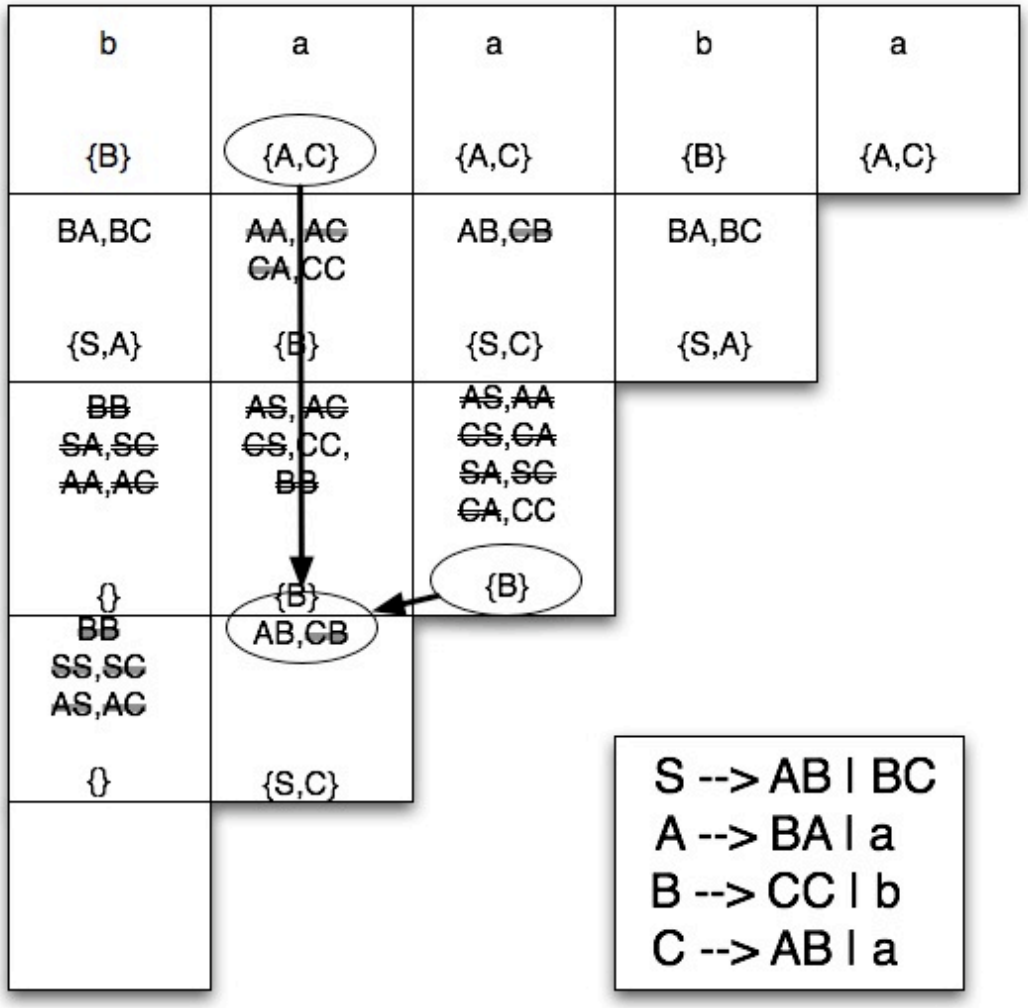
CYK Parser (Sequential Version)



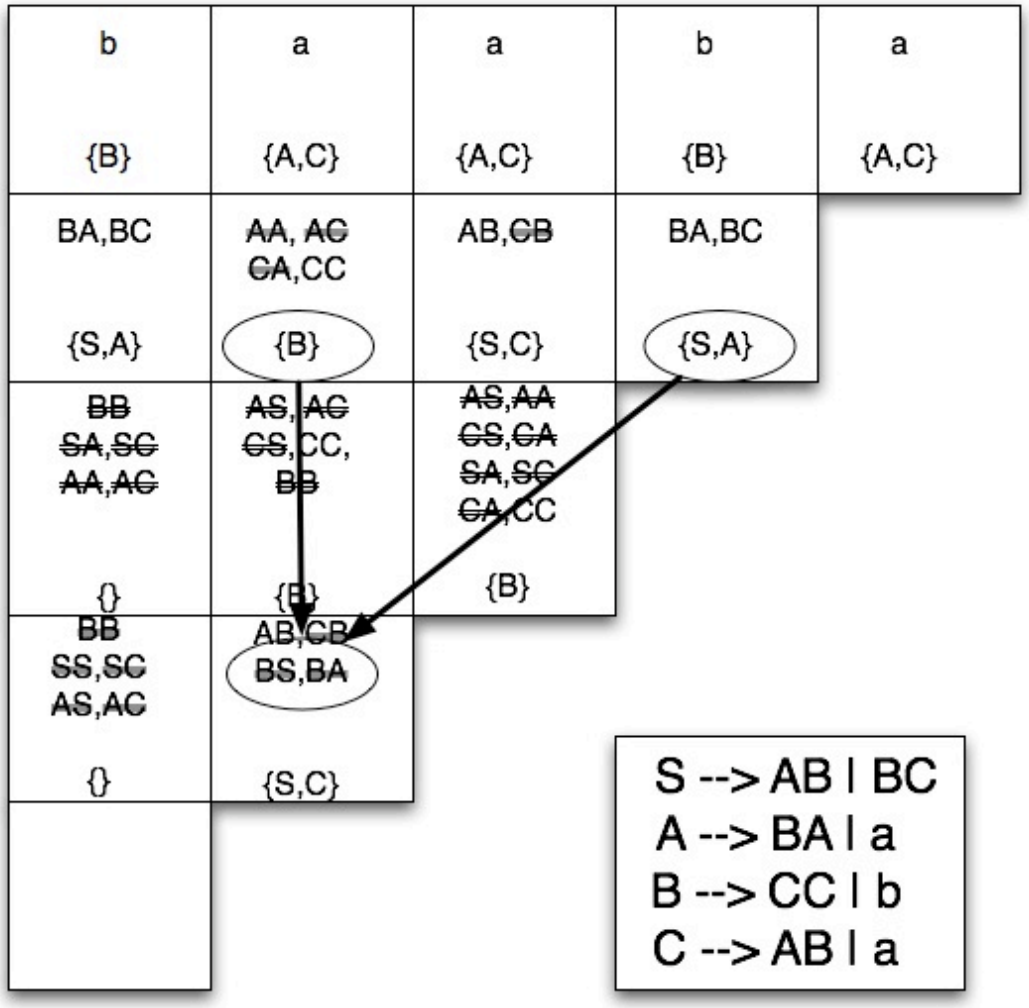
CYK Parser (Sequential Version)



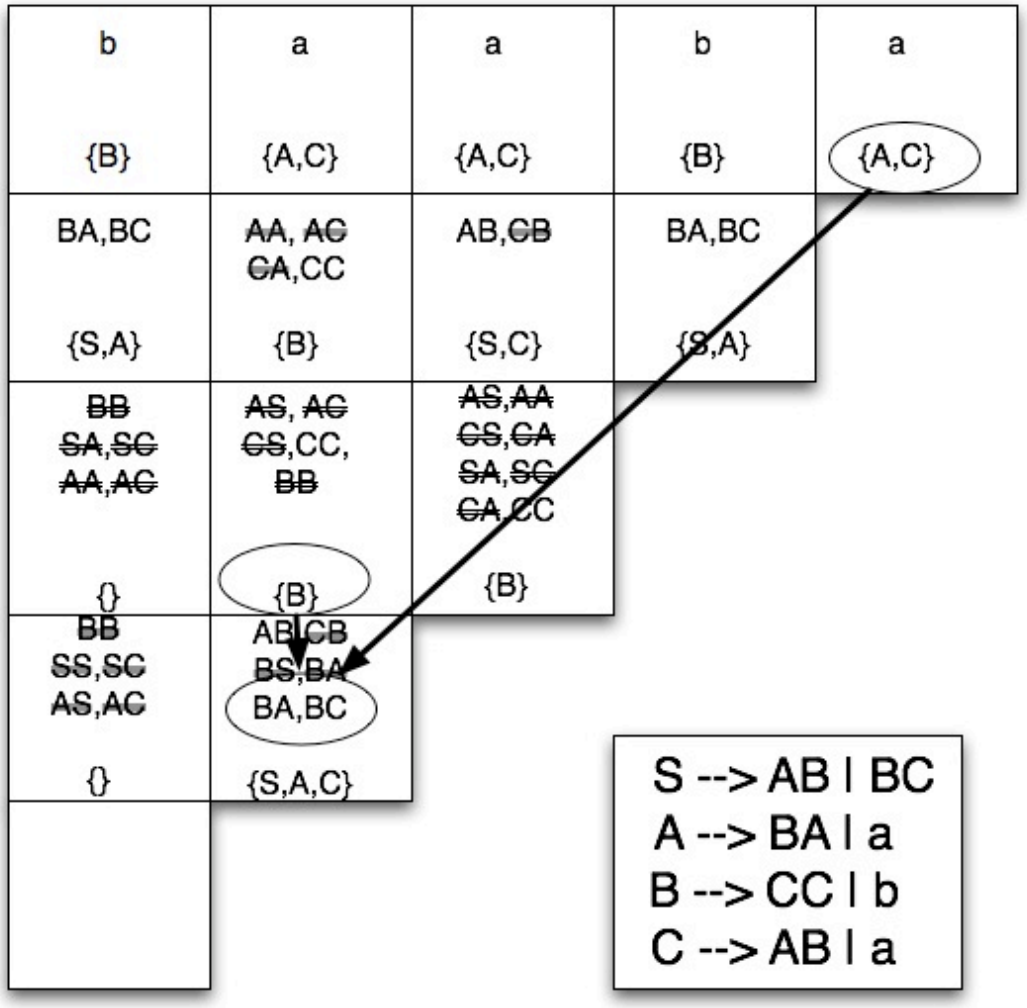
CYK Parser (Sequential Version)



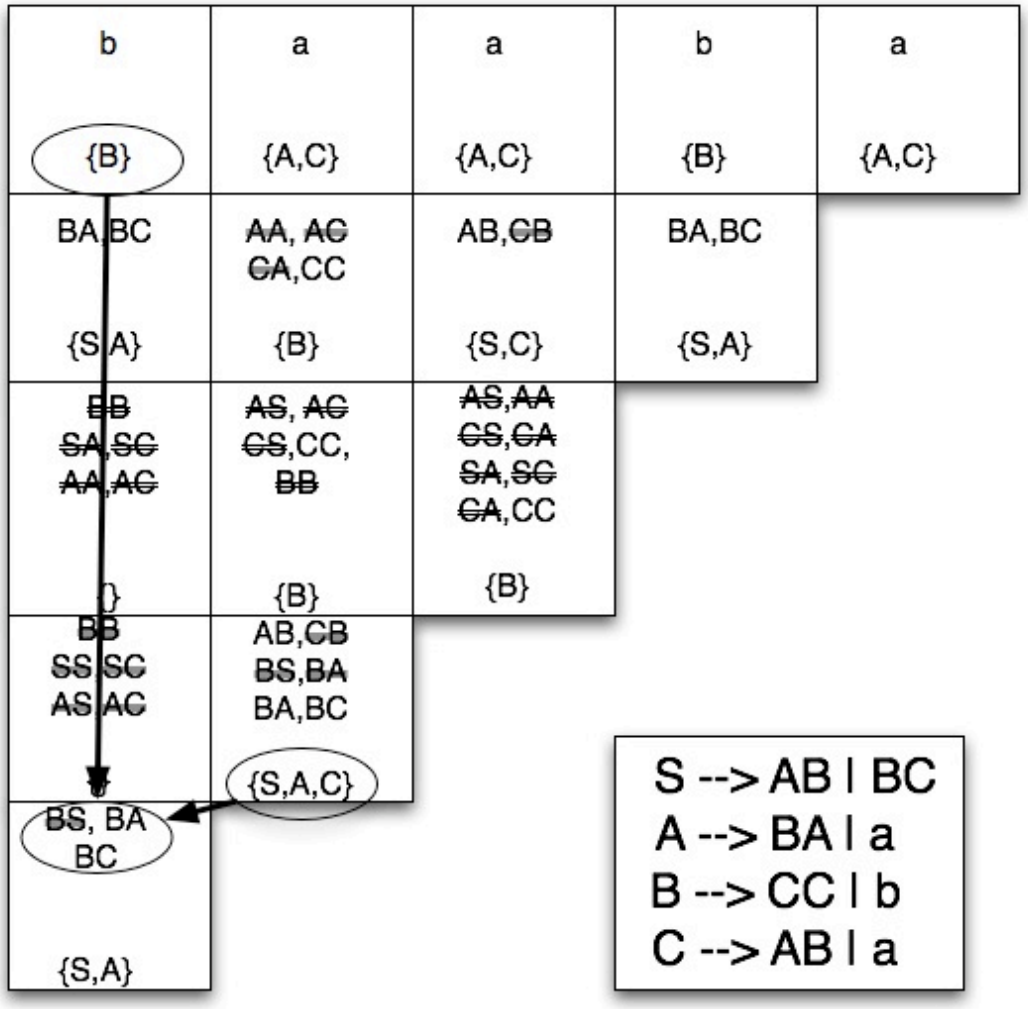
CYK Parser (Sequential Version)



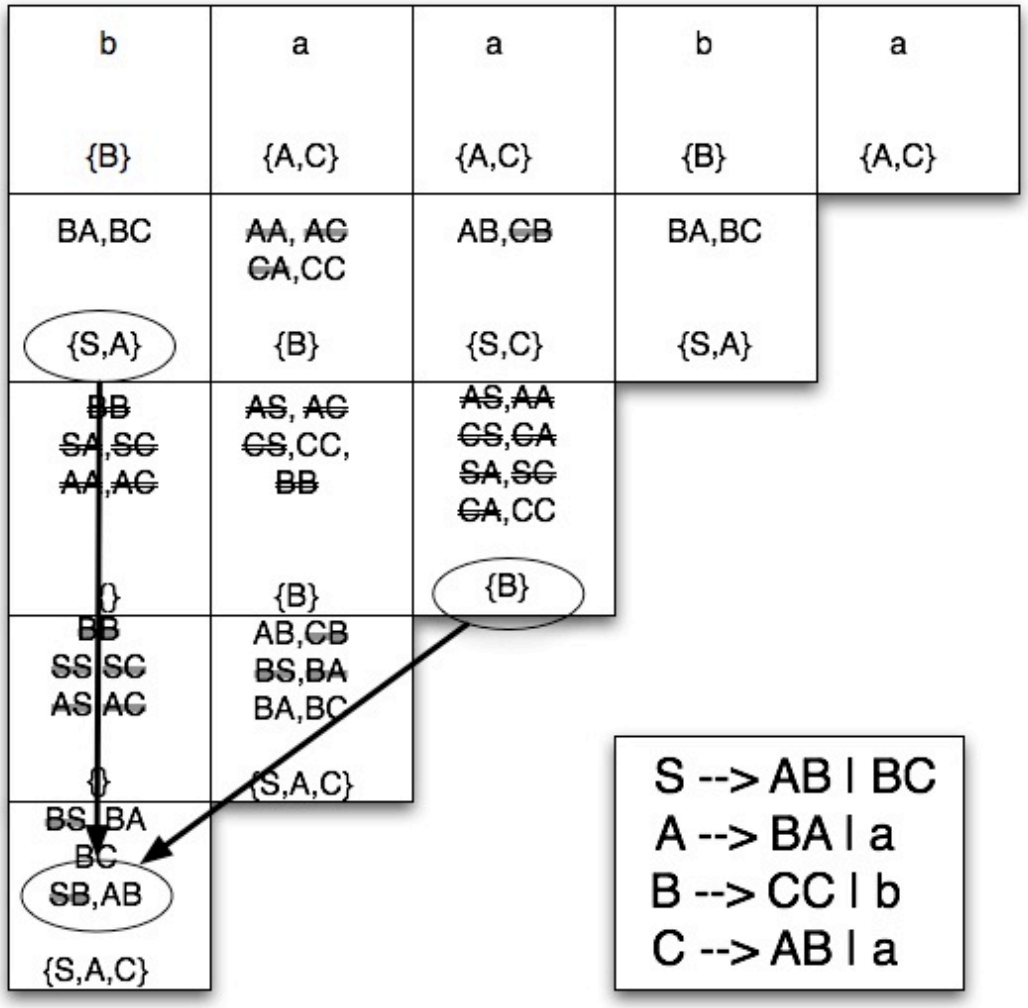
CYK Parser (Sequential Version)



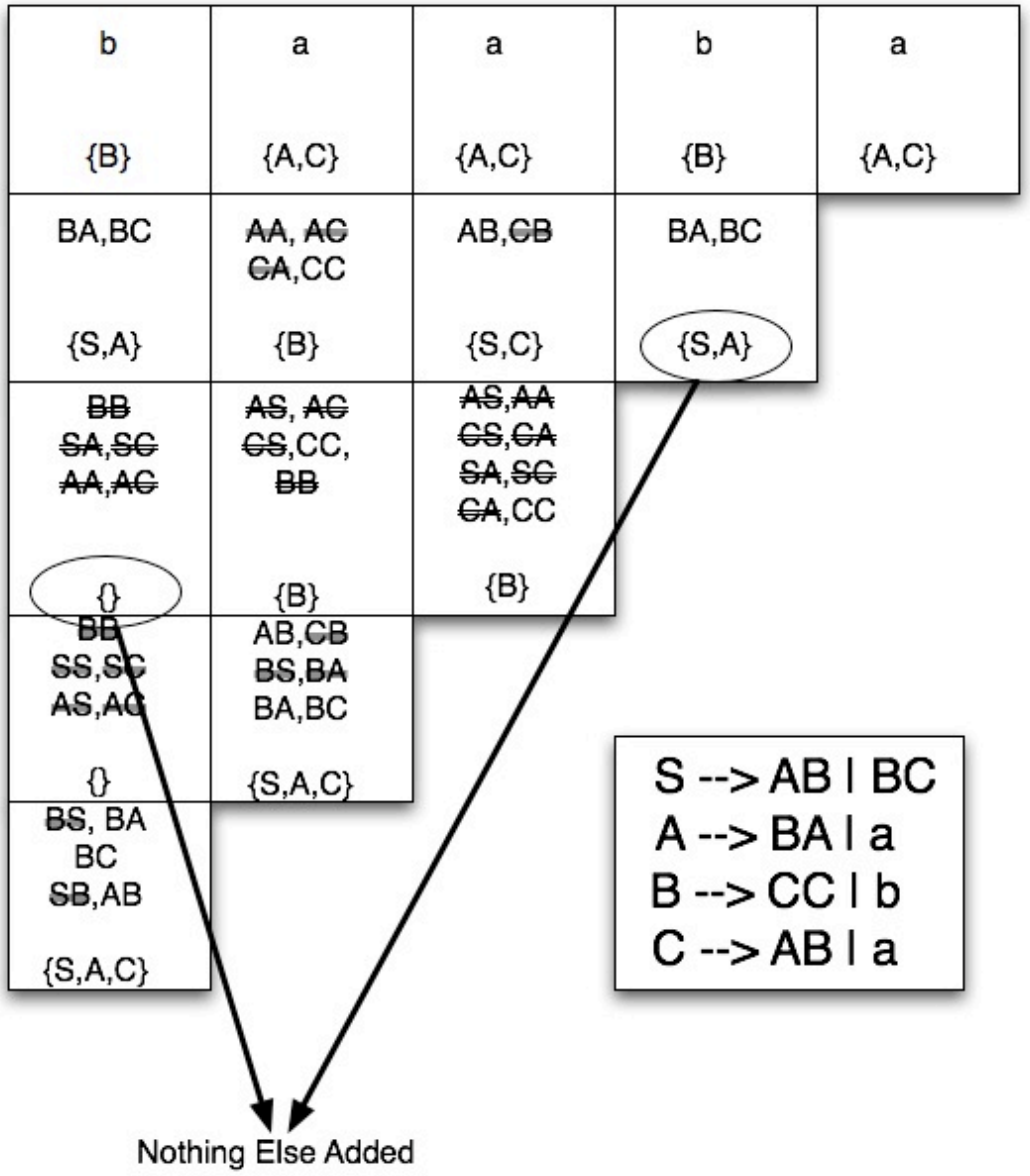
CYK Parser (Sequential Version)



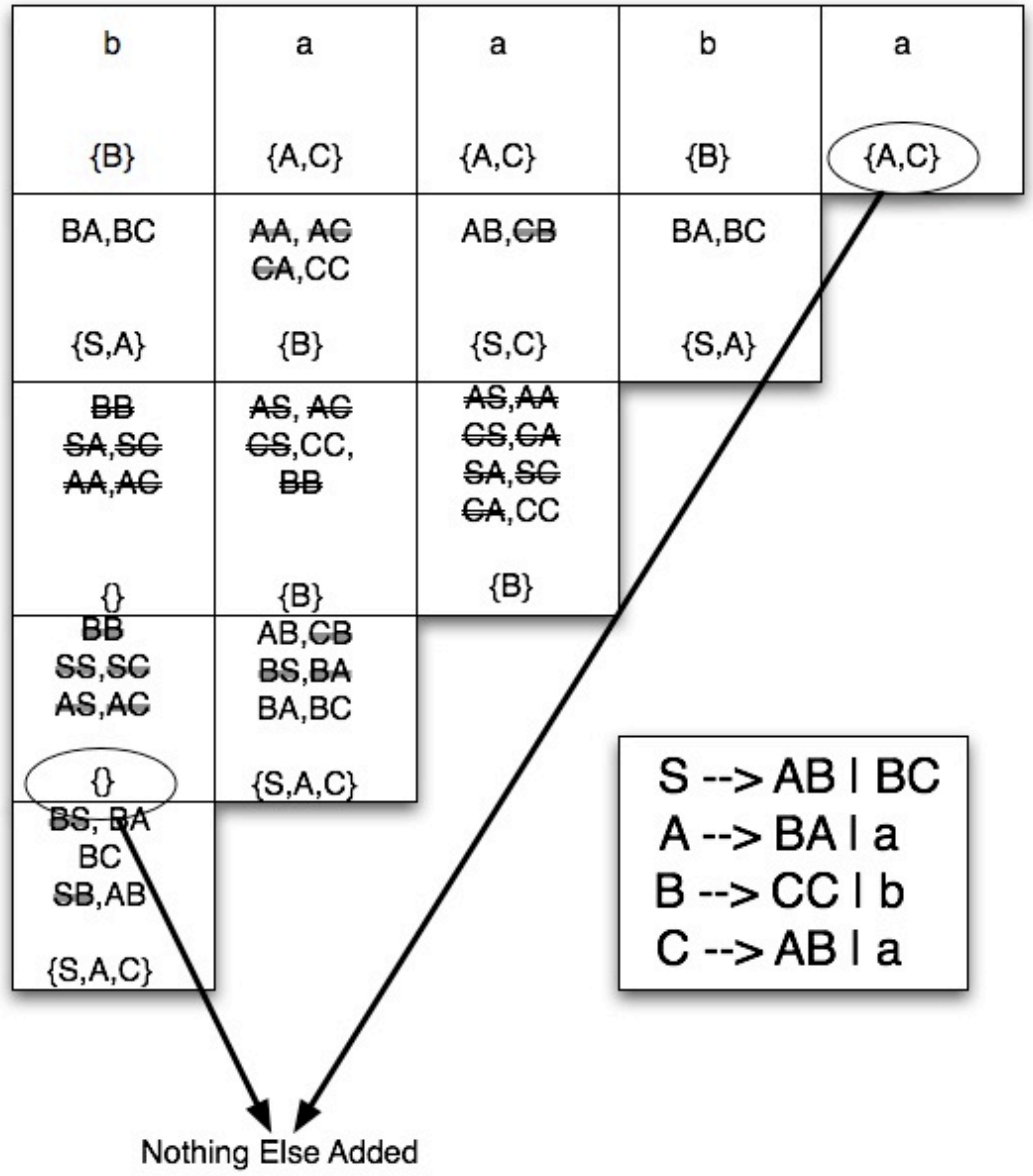
CYK Parser (Sequential Version)



CYK Parser (Sequential Version)



CYK Parser (Sequential Version)



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

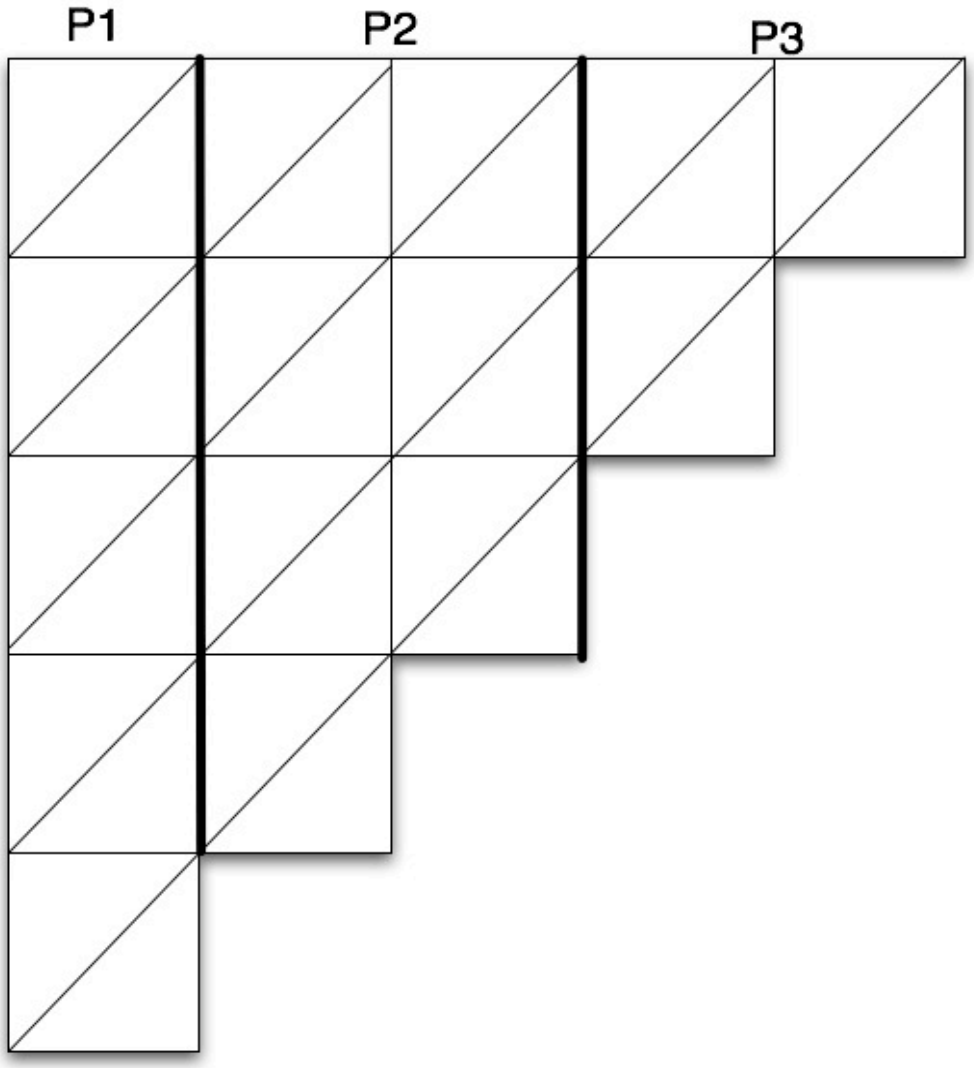
The CYK Parser Algorithm (Sequential Version)

```
(* 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  }
```

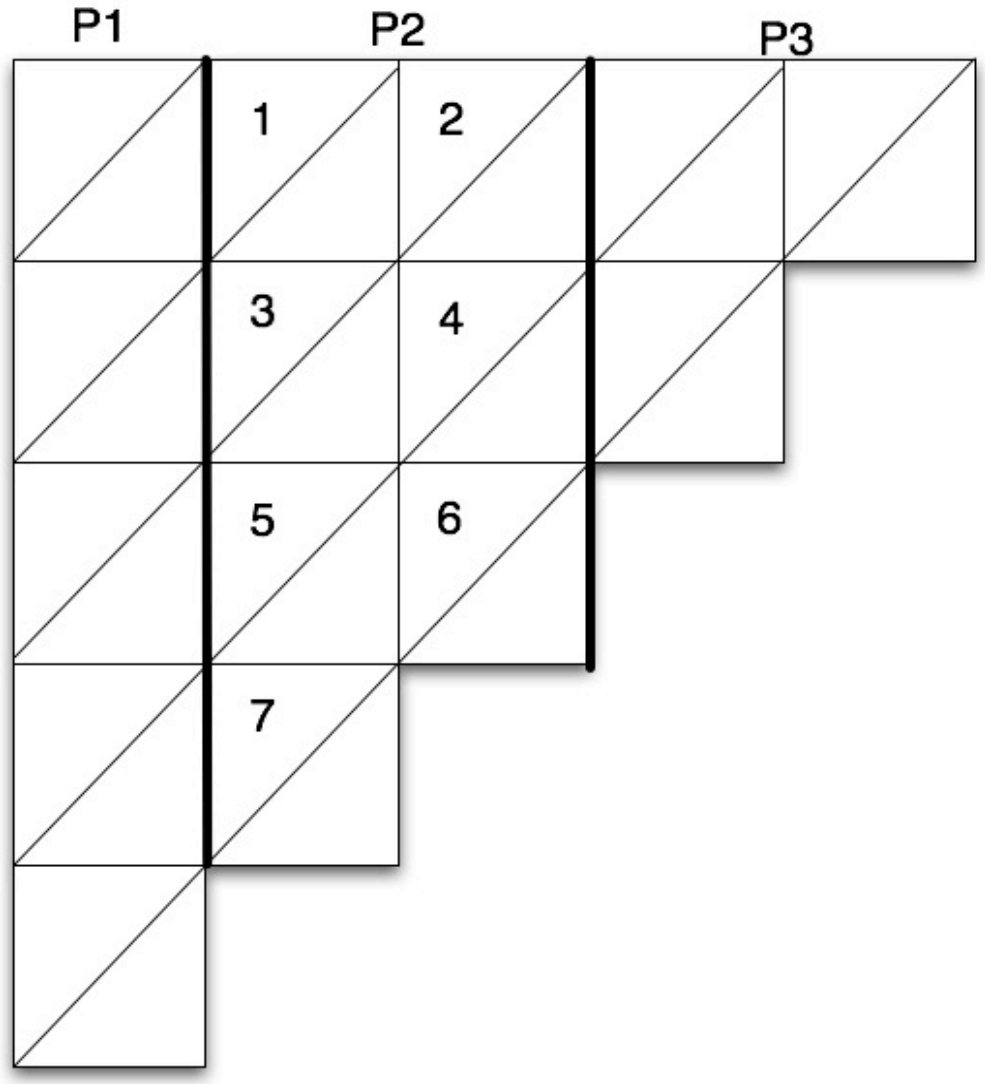
Figure3: Pseudo-code for the sequential CYK algorithm. Adapted from Hopcroft, Ullman, 1979, pp139-140.

CYK Parser (Parallel Version)



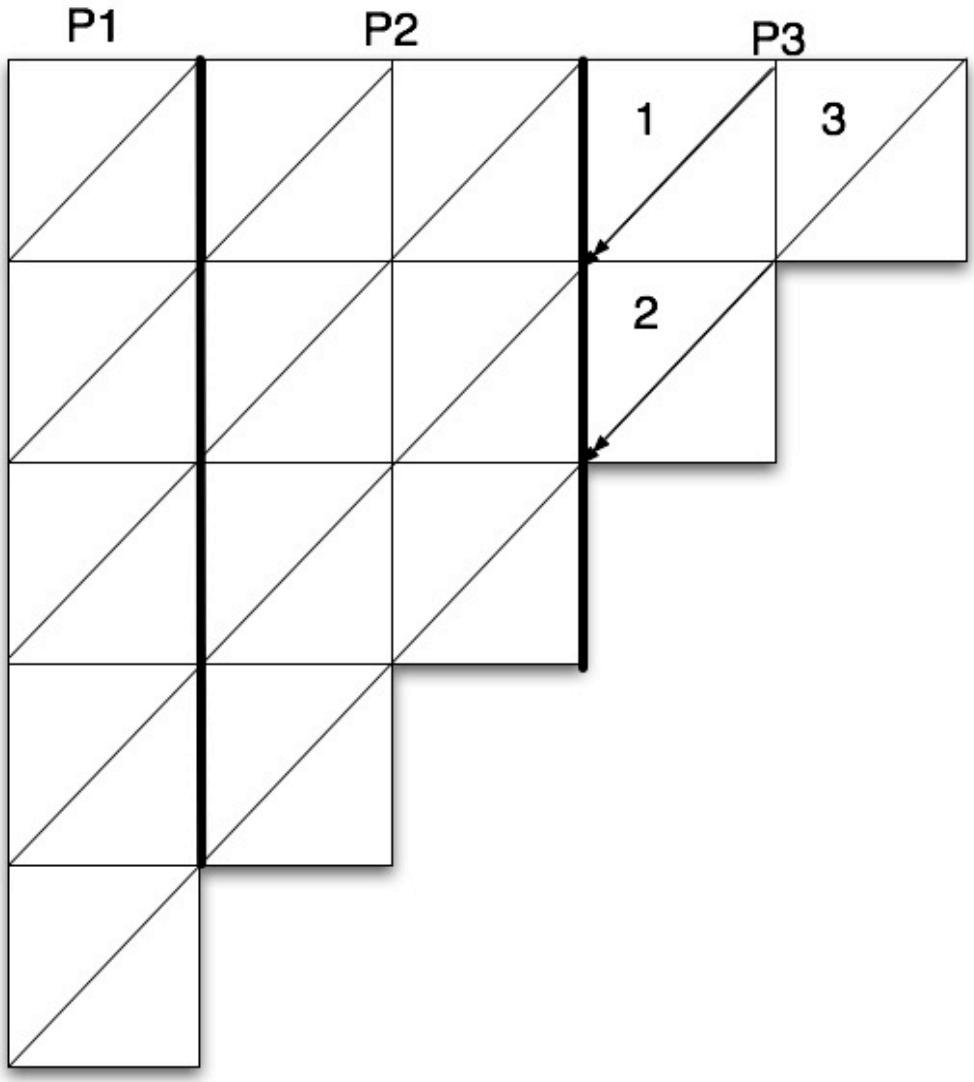
Matrix for a string of length 5 using 3 processors

CYK Parser (Parallel Version)



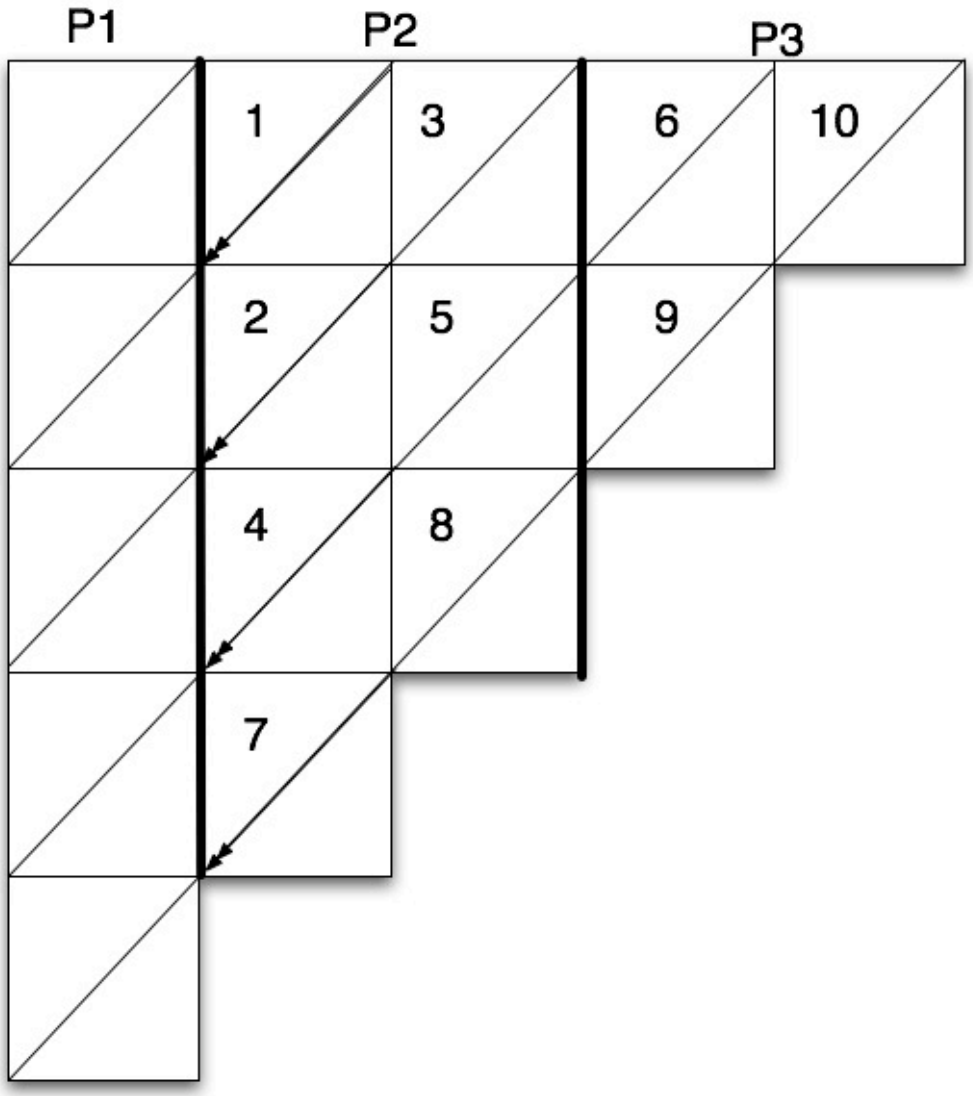
Order of calculation for processor P2. P2 calculates a diagonal at a time.

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

Extra Slides Start Here

Example (*grammar & sets*)

Simplified, right recursive expression grammar

Goal → *Expr*
Expr → *Term* - *Expr*
Expr → *Term*
Term → *Factor* * *Term*
Term → *Factor*
Factor → *ident*

Symbol	FIRST
Goal	{ ident }
Expr	{ ident }
Term	{ ident }
Factor	{ ident }
-	{ - }
*	{ * }
ident	{ ident }

Example (*building the collection*)

Initialization Step

$s_0 \leftarrow$ *closure(* { [*Goal* → • *Expr* , EOF] } *)*

{ [*Goal* → • *Expr* , EOF], [*Expr* → • *Term* – *Expr* , EOF],
[*Expr* → • *Term* , EOF], [*Term* → • *Factor* * Term , EOF],
[*Term* → • *Factor* * *Term* , –], [*Term* → • *Factor* , EOF],
[*Term* → • *Factor* , –], [*Factor* → • ident , EOF],
[*Factor* → • ident , –], [*Factor* → • ident , *] }

S $\leftarrow \{s_0\}$

Example (*building the collection*)

Iteration 1

$s_1 \leftarrow goto(s_0 , Expr)$

$s_2 \leftarrow goto(s_0 , Term)$

$s_3 \leftarrow goto(s_0 , Factor)$

$s_4 \leftarrow goto(s_0 , \underline{ident})$

Iteration 2

$s_5 \leftarrow goto(s_2 , -)$

$s_6 \leftarrow goto(s_3 , *)$

Iteration 3

$s_7 \leftarrow goto(s_5 , Expr)$

$s_8 \leftarrow goto(s_6 , Term)$

Example (*Summary*)

S_0 : { [*Goal* → • *Expr* , EOF], [*Expr* → • *Term* – *Expr* , EOF],
[*Expr* → • *Term* , EOF], [*Term* → • *Factor* * *Term* , EOF],
[*Term* → • *Factor* * *Term* , –], [*Term* → • *Factor* , EOF],
[*Term* → • *Factor* , –], [*Factor* → • ident , EOF],
[*Factor* → • ident , –], [*Factor*→ • ident, *] }

S_1 : { [*Goal* → *Expr* •, EOF] }

S_2 : { [*Expr* → *Term* • – *Expr* , EOF], [*Expr* → *Term* •, EOF] }

S_3 : { [*Term* → *Factor* • * *Term* , EOF],[*Term* → *Factor* • * *Term* , –],
[*Term* → *Factor* •, EOF], [*Term* → *Factor* •, –] }

S_4 : { [*Factor* → ident •, EOF],[*Factor* → ident •, –], [*Factor* → ident •, *] }

S_5 : { [*Expr* → *Term* – • *Expr* , EOF], [*Expr* → • *Term* – *Expr* , EOF],
[*Expr* → • *Term* , EOF], [*Term* → • *Factor* * *Term* , –],
[*Term* → • *Factor* , –], [*Term* → • *Factor* * *Term* , EOF],
[*Term* → • *Factor* , EOF], [*Factor* → • ident , *],
[*Factor* → • ident , –], [*Factor* → • ident , EOF] }

Example (*Summary*)

S_6 : { [*Term* → *Factor* * • *Term* , EOF], [*Term* → *Factor* * • *Term* , –],
[*Term* → • *Factor* * *Term* , EOF], [*Term* → • *Factor* * *Term* , –],
[*Term* → • *Factor* , EOF], [*Term* → • *Factor* , –],
[*Factor* → • ident , EOF], [*Factor* → • ident , –], [*Factor* → • ident , *] }

S_7: { [*Expr* → *Term* – *Expr* •, EOF] }

S_8 : { [*Term* → *Factor* * *Term* •, EOF], [*Term* → *Factor* * *Term* •, –] }

Example (*Summary*)

The Goto Relationship *(from the construction)*

State	Expr	Term	Factor	-	*	Ident
0	1	2	3			4
1						
2				5		
3					6	
4						
5	7	2	3			4
6		8	3			4
7						
8						

Filling in the ACTION and GOTO Tables

The algorithm

x is the number of the state for s_x

∀ *set* $s_x \in S$
  ∀ *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* $[S' \rightarrow S \cdot, EOF]$
      then ACTION$[x, EOF] \leftarrow$ "*accept*"
    else if *i is* $[A \rightarrow \beta \cdot, \underline{a}]$
      then ACTION$[x, \underline{a}] \leftarrow$ "*reduce* $A \rightarrow \beta$"
  ∀ $n \in NT$
    if $goto(s_x, n) = s_k$
      then GOTO$[x, n] \leftarrow k$

Many items generate no table entry

e.g., $[A \rightarrow \beta \cdot B \alpha, \underline{a}]$ does not, but *closure* ensures that all the rhs' for *B* are in s_x

Example (*Filling in the tables*)

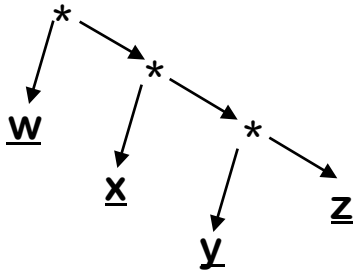
The algorithm produces the following table

	ACTION				GOTO		
	Ident	-	*	EOF	*Expr*	*Term*	*Factor*
0	s 4				1	2	3
1				acc			
2		s 5		r 3			
3		r 5	s 6	r 5			
4		r 6	r 6	r 6			
5	s 4				7	2	3
6	s 4					8	3
7				r 2			
8		r 4		r 4			

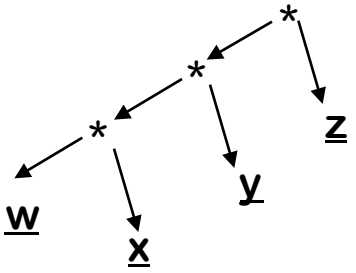
Plugs into the skeleton LR(1) parser

Left Recursion versus Right Recursion

- Right recursion
- Required for termination in top-down parsers
- Uses (on average) more stack space
- Produces right-associative operators
- Left recursion
- Works fine in bottom-up parsers
- Limits required stack space
- Produces left-associative operators
- Rule of thumb
- Left recursion for bottom-up parsers
- Right recursion for top-down parsers



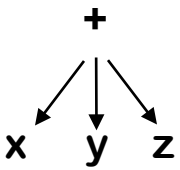
w * (x * (y * z))



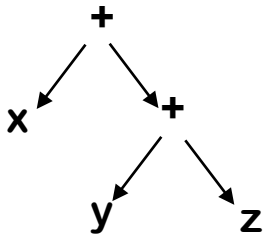
((w * x) * y) * z

Associativity

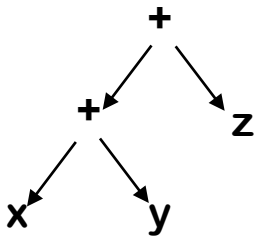
- What difference does it make?
- Can change answers in floating-point arithmetic
- Exposes a different set of common subexpressions
- Consider x+y+z



Ideal operator



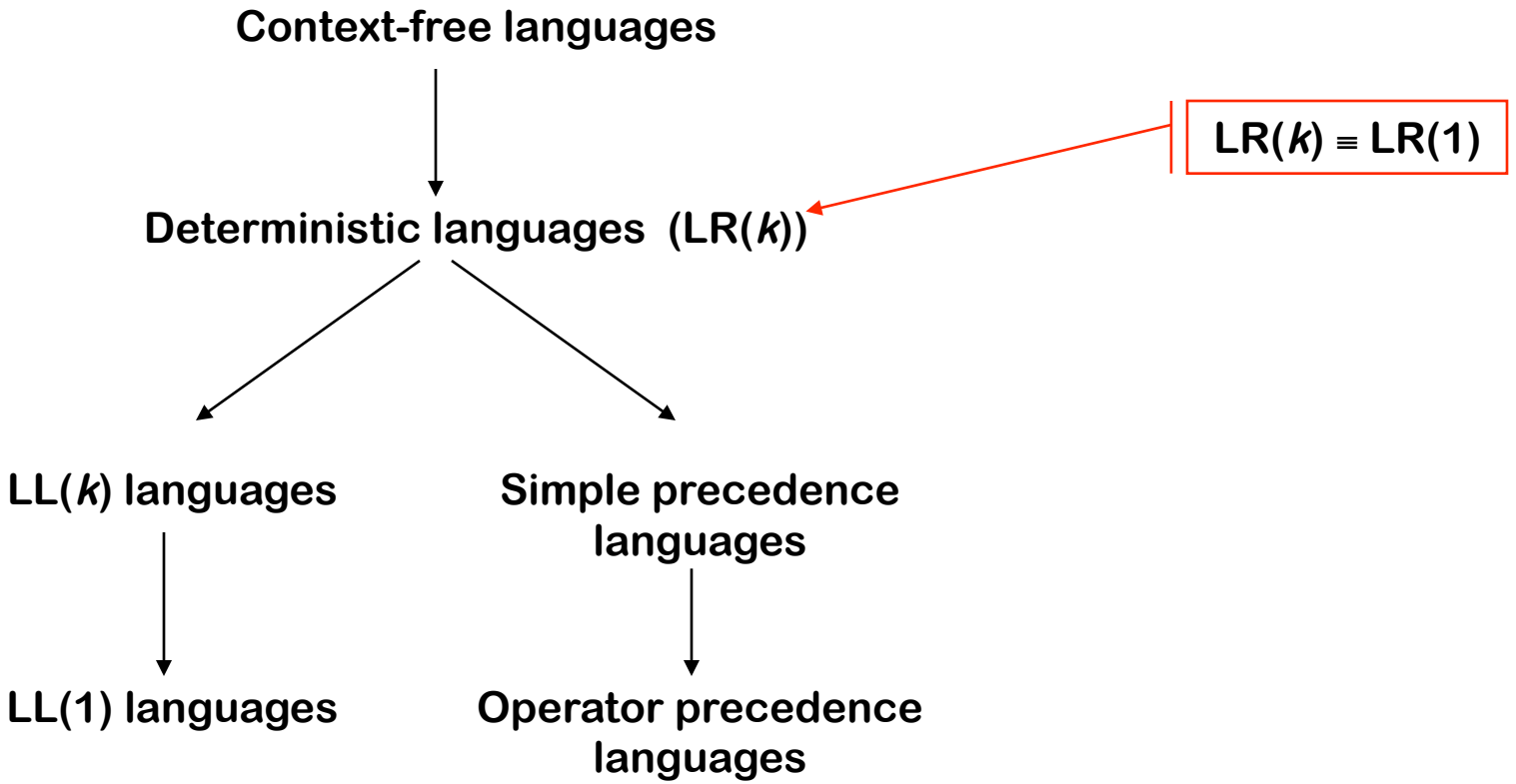
Left association



Right association

- What if y+z occurs elsewhere? Or x+y? or x+z?
- What if x = 2 & z = 17 ? Neither left nor right exposes 19.
- Best choice is function of surrounding context

Hierarchy of Context-Free Languages



The inclusion hierarchy for context-free languages

Beyond Syntax

There is a level of correctness that is deeper than grammar

```
fie(a,b,c,d)
   int a, b, c, d;
{ … }
fee() {
   int f[3],g[0],
     h, i, j, k;
  char *p;
   fie(h,i,"ab",j, k);
   k = f * i + j;
   h = g[17];
   printf("<%s,%s>.\n",
     p,q);
   p = 10;
}
```

What is wrong with this program?

(let me count the ways …)

To generate code, we need to understand its meaning !

There is a level of correctness that is deeper than grammar

```
fie(a,b,c,d)
   int a, b, c, d;
{ … }
fee() {
   int f[3],g[0],
     h, i, j, k;
  char *p;
   fie(h,i,"ab",j, k);
   k = f * i + j;
   h = g[17];
   printf("<%s,%s>.\n",
     p,q);
   p = 10;
}
```

What is wrong with this program?

(let me count the ways …)

- declared g[0], used g[17]
- wrong number of args to fie()
- "ab" is not an int
- wrong dimension on use of f
- undeclared variable q
- 10 is not a character string

All of these are "deeper than syntax"

Beyond Syntax

To generate code, the compiler needs to answer many questions

- Is "x" a scalar, an array, or a function? Is "x" declared?
- Are there names that are not declared? Declared but not used?
- Which declaration of "x" does each use reference?
- Is the expression "x * y + z" type-consistent?
- In "*a*[i,j,k]", does *a* have three dimensions?
- Where can "z" be stored? *(register, local, global, heap, static)*
- In "f ← 15", how should 15 be represented?
- How many arguments does "fie()" take? What about "printf ()" ?
- Does "*p" reference the result of a "malloc()" ?
- Do "p" & "q" refer to the same memory location?
- Is "x" defined before it is used?

These cannot be expressed in a CFG

Beyond Syntax

These questions are part of context-sensitive analysis

- Answers depend on values, not parts of speech
- Questions & answers involve non-local information
- Answers may involve computation

How can we answer these questions?

- Use formal methods
  - → Context-sensitive grammars?
  - → Attribute grammars? *(attributed grammars?)*
- Use *ad-hoc* techniques
  - → Symbol tables
  - → *Ad-hoc* code *(action routines)*

In scanning & parsing, formalism won; different story here.

Beyond Syntax

Telling the story

- The attribute grammar formalism is important
  - → Succinctly makes many points clear
  - → Sets the stage for actual, *ad-hoc* practice
- The problems with attribute grammars motivate practice
  - → Non-local computation
  - → Need for centralized information
- Some folks in the community still argue for attribute grammars
  - → Knowledge is power
  - → Information is immunization

We will cover attribute grammars, then move on to *ad-hoc* ideas