Language Parsing

Robert M. Keller
Harvey Mudd College
April 2013

Parsing

- Parsing is the process of determining whether a string is in a given language.
- Every compiler and interpreter does some form of parsing.
- Closely related is the process of assigning a meaning to the strings that are in the language.
- Refer to CS 60 materials for a review of how this can be done.

Parsing with a Pushdown Automaton

- We have seen how pushdown automata parse languages represented by context-free grammars.
- In some cases, the corresponding automaton is non-deterministic, which limits the practical use of pda's as a programming technique.
- Only a proper subset of languages can be parsed **deterministically** by a pda. Intuitively, these Deterministic Context-Free Languages are the ones that don't require "guessing".

DPDAs (Deterministic PDAs)

- PDAs are non-deterministic by default.
- A DPDA restricts transitions so that the PDA behaves deterministically.
- ε transitions are still allowed, with restrictions.

DPDAs

- δ: Q x $\Sigma\varepsilon$ x $\Gamma\varepsilon \rightarrow$ Q x $\Gamma\varepsilon$
  (rather than to subsets of Q x $\Gamma\varepsilon$)

- Exactly one of these combinations is allowed for any (q, σ, γ):
  - δ(q, σ, γ) where $\sigma \neq \varepsilon$ and $\gamma \neq \varepsilon$
  - δ(q, ε, γ) where $\gamma \neq \varepsilon$
  - δ(q, σ, ε) where $\sigma \neq \varepsilon$
  - δ(q, ε, ε)

Pathological Behaviors

- Even with the given restrictions, the following behaviors could happen:
  - In a given machine state, there could be infinite cycling of ε moves where no further input symbol is read.
  - In a given machine state, the stack is empty, yet further reading can only take place if the stack is non-empty.
- Sipser Lemma 2.41 says that any DPDA can be transformed into one that reads the entire input (leaving it in an accepting or rejecting state).

DCFLs are closed under complementation

- This is Sipser Theorem 2.42
- The proof requires removal of a further pathology: A state cannot be an accept state unless it is “trying to read”. In other words, it is not a state with an epsilon transition leaving.

- I will not hold you responsible for proving this theorem and the previous lemma. But it’s good to know that these issues lurk and can be solved.

A Sufficient Condition *not* to be DCFL

- DFCLs are closed under complementation, but we know CFLs are not in general.
- Thus if L is a CFL but its complement is not, then L cannot be a DCFL.
- Example: $\{a^i b^j c^k \mid i \neq j \vee j \neq k\}$ is not a DFCL.
- The complement of L, intersected with a*b*c* (which is regular) is $\{a^n b^n c^n \mid n \geq 0\}$, which is not CF.

Use of End-Markers

- In general, a pda will tell us when to accept the string parsed **so far**, as if there might be more coming.
- Sometimes we want to tell the pda that we **don't want an answer** to this question until all input has been supplied.
- One way to do this is to include an **endmarker**, say $, in the language being defined. The language will have the form

  {x$ | x has some property}
- The pda can react to the endmarker when it sees it.

Deterministic Parsing using Look-ahead

- A non-deterministic pda can sometimes be rendered “deterministic” if there is some way to restrict the choice of moves to at most one.

- The best-known cases are to use “look-ahead”:
  - By using **knowledge of the next symbol in the input**, only certain rules (corresponding to certain grammar productions) can be seen to be applicable. The ability to do this depends on having the appropriate grammar. **The knowledge itself is implicit in the grammar**.

Deterministic Parsing using Look-ahead

- The best-known cases are to use “look-ahead”:
  - The LR(1) grammars can be parsed **bottom-up** using **shift-reduce** corresponding to a right-most one symbol look-ahead.
  - The LL(1) grammars can be parsed **top-down** using **produce-match** corresponding to a left-most derivation with one symbol look-ahead.
  - The concept of LL(k) and LR(k) are due to Donald Knuth.

LR(1)

- Handles a broader family of languages than LL(1) can.
- Difficult to transform an arbitrary grammar to this form.
- Automated tools such as YACC can help.
- See section 2.2 of Sipser 3rd Edition on DCFLs.
- I won't hold you responsible for knowing.

LL(1)

- Top-Down Recursive-Descent
- Deterministic Produce-Match Parsing
- Use 1-symbol look-ahead to decide which production to apply.

Eliminate Left-Recursion First

- Consider productions for a comma-separated list:

  $L \rightarrow L\ ,\ I \ \mid \ I$

- Will loop infinitely top-down.

- Instead add a new non-terminal, M and use:

  $L \rightarrow I\ M$

  $M \rightarrow ,\ I\ M \mid \varepsilon$

Without Left-Recursion

- L → a M

  M → , a M | ε

- Suppose the input is a,a,a

- Is it always clear which production to apply, just knowing the next symbol in input?

LL(1) Parsing Table

- This is a control mechanism to indicate what production should be applied given:
  - The next symbol in the input
  - The symbol on top of the stack
- Such a table can be **computed** from an LL (1) grammar.
- If the computation fails, the grammar is not LL(1).

When to use a production $X \rightarrow \gamma$

- If σ is the next symbol in the input and X is on top of the stack, then use $X \rightarrow \gamma$ when either:
  - $\gamma \Rightarrow^* \sigma x$ for some x, or
  - $\gamma \Rightarrow^* \varepsilon$ and σ is the first symbol in some string that can follow X.
- These conditions are captured by:
  - $\sigma \in \text{First}(\gamma)$, or
  - $\sigma \in \text{Follow}(X)$ and γ is "nullable"

First, Define Nullable

- A string x is **nullable** provided x $\Rightarrow^* \varepsilon$.
- We are interested in nullability for left- and right-hand sides of productions.
- Nullability can be computed by **fixed-point iteration** over the grammar:
  - If X $\rightarrow \varepsilon$ then X is nullable.
  - If X $\rightarrow$ AB…C and each of A, B, C are nullable, then AB… C and X are nullable.
  - The only nullable production elements are those that can be determined by applying the above rules.

Nullability Computation Example

$Z \rightarrow X\ Y$	$Y \rightarrow c$	$X \rightarrow a$	$W \rightarrow aX$
$Z \rightarrow d$	$Y \rightarrow \varepsilon$	$X \rightarrow Y$	$W \rightarrow Yb$

- Y is nullable by $Y \rightarrow \varepsilon$
- X is nullable by $X \rightarrow Y$
- Z is nullable by $Z \rightarrow X\ Y$
- W is not nullable

Define First(x) where $x \in (\Sigma \cup N)^*$

- Definition: For any $x \in (\Sigma \cup N)^*$

  $First(x) = \{\sigma \in \Sigma \mid \exists y \in (\Sigma \cup N)^* \; x \Rightarrow^* \sigma y\}$

- In other words, $First(x) \subseteq \Sigma$ such that x derives a string beginning with $\sigma \in \Sigma$.

How we use *First*

- Suppose X is a non-terminal on top of the stack and σ is the next character in the input.
- In order for the top-down parse to succeed at this point, $\sigma \in$ First(X) is required.
- So of possibly several right-hand sides $X \rightarrow \gamma$, $\gamma \in (\Sigma \cup N)^*$, if there is a **unique** $X \rightarrow \gamma$ having $\sigma \in$ First(γ), we should replace X with γ on the stack.

Computing *First*

- Our main interest is First(X) for non-terminals X and First(γ) for γ in productions X $\rightarrow$ γ.

- It is easier to describe how to compute than it is to write the definition out as sets, so let's use a computational definition.

Computing First (after Rodger & Finley: JFLAP book)

By recursion:

- First(σ) = $\{\sigma\}$ for each $\sigma \in \Sigma$
- First(X) = $\cup\{$First(γ) | X $\rightarrow \gamma\}$ for each X $\in$ N
- For right-hand sides:
  - First(AB…C) = First(A) if A is not nullable
  - First(AB…C) = First(A) $\cup$ First(B…C) if A is nullable

Example *First* Computation for LHS's

$Z \rightarrow X\ Y\ Z$
$Z \rightarrow d$
$Z \rightarrow Y\ f$

$Y \rightarrow c$
$Y \rightarrow \varepsilon$

$X \rightarrow a$
$X \rightarrow b\ Y\ e$

- First(X) = {a, b}
- First(Y) = {c}
- First(Z) = First(X) ∪ {d} ∪ First(Y) ∪ {f}
  = {a, b, c, d, f}
- (Y is the only nullable non-terminal)

Use of First for Produce Moves

- **In the absence of nullable non-terminals**, using First is adequate to **predict** which production to use.
- If X is on the stack, σ is the next input character, then we should have $\sigma \in$ First(X) to succeed.
- If there is a **unique** $X \rightarrow \gamma$ with $\sigma \in$ First(γ),
  we know that we should **replace X with that** γ as the next move of the PDA.
- If γ begins with σ, we combine the match with σ in this step.

Pure Match Moves

- If σ is on top the stack and σ is the next input character, then we should pop and move on.

Example without Nullables

- S → aA | bB
- A → aB | B
- B → bA | c

Input: aabbbac	Stack: S	Apply: S → aA
Input: abbbac	Stack: A	Apply: A → aB
Input: bbbac	Stack: B	Apply: B → bA
Input: bbac	Stack: A	Apply: A → B
Input: bbac	Stack: B	Apply: B → bA
Input: bac	Stack: A	Apply: A → B
Input: bac	Stack: B	Apply: B → bA
Input: ac	Stack: A	Apply: A → aB
Input: c	Stack: B	Apply: B → c
Input: ε	Stack: ε	**accept**

When Nullable Non-Terminals are Present

- If a nullable non-terminal X is on top, then it is possible to have **no** $X \rightarrow \gamma$ with the next input symbol $\sigma \in \text{First}(\gamma)$, **and still succeed**.
- This is because $X \Rightarrow^* \varepsilon$ could lead to **exposure** of the symbol Y below X in the stack for which $\sigma \in \text{First}(Y)$.
- **Follow(X)**, defined next, tells when this is the case.

Informal Definition of Follow

- For a non-terminal X, Follow(X) is the set of terminals that could follow X in some string during a left-most derivation.

Follow(x) where x $\in \Sigma \cup N$

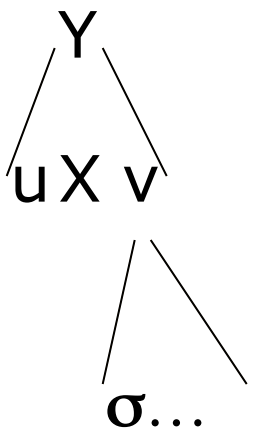
- Follow is computed by fixed-point iteration, after First has been computed.
- Initially Follow(X) = {}.

A. If there is a production Y $\rightarrow$ uXv, then include elements of First(v) in Follow(X).

B. If Y $\rightarrow$ uXAB…C, and A, B, …, C (0 or more) are nullable, then include elements of Follow(Y) within Follow(X).

A. Justification

- If there is a production $Y \rightarrow uXv$, then include First(v) in Follow(X).

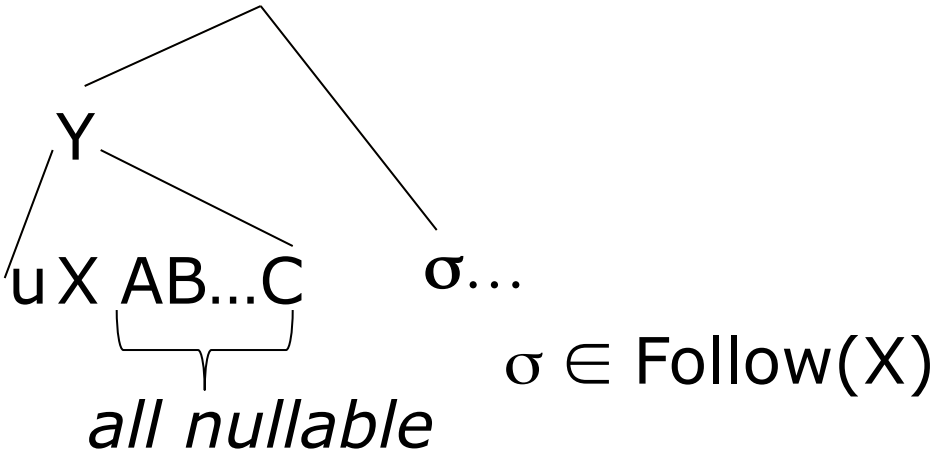


$\sigma \in$ First(v)

i.e. σ can follow X.

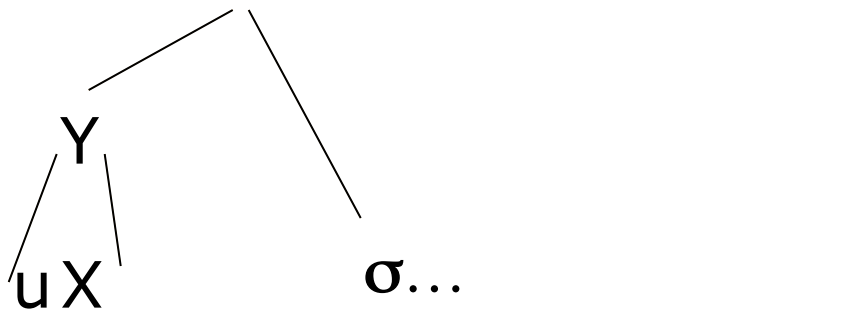
B. Justification

- If Y → uXAB…C, and A, B, …, C (0 or more) are nullable, then include
  Follow(Y) ⊆ Follow(X).



Special Case of B.

- If $Y \rightarrow uX$, then include
  $\text{Follow}(Y) \subseteq \text{Follow}(X)$.



$\sigma \in \text{Follow}(X)$

Example of Parse Table Construction

Example adapted from David Walker at UCB.

Z → X Y Z
Z → d
Y → c
Y → ε
X → a
X → b Y e

	nullable	first	follow
Z	no		
Y	yes		
X	no		

Compute First

$Z \rightarrow X\ Y\ Z$ $\quad$ $Y \rightarrow c$ $\quad$ $X \rightarrow a$

$Z \rightarrow d$ $\quad$ $Y \rightarrow \varepsilon$ $\quad$ $X \rightarrow b\ Y\ e$

First iteration of first:

	nullable	first	follow
Z	no	d	
Y	yes	c	
X	no	a,b	

Computing First

$Z \rightarrow X\ Y\ Z$ $\quad$ $Y \rightarrow c$ $\quad$ $X \rightarrow a$

$Z \rightarrow d$ $\quad$ $Y \rightarrow \varepsilon$ $\quad$ $X \rightarrow b\ Y\ e$

Second iteration of first:

	nullable	first	follow
Z	no	a,b,d	
Y	yes	c	
X	no	a,b	

Computing First

$Z \rightarrow X\ Y\ Z$ $\quad$ $Y \rightarrow c$ $\quad$ $X \rightarrow a$

$Z \rightarrow d$ $\quad$ $Y \rightarrow \varepsilon$ $\quad$ $X \rightarrow b\ Y\ e$

Closure: First is complete.

	nullable	first	follow
Z	no	a,b,d	
Y	yes	c	
X	no	a,b	

Compute Follow

$Z \rightarrow X\ Y\ Z$ $\quad$ $Y \rightarrow c$ $\quad$ $X \rightarrow a$

$Z \rightarrow d$ $\quad$ $Y \rightarrow \varepsilon$ $\quad$ $X \rightarrow b\ Y\ e$

A. From $Z \rightarrow X\ Y\ Z$ and first(Z) = {a,b,d},
add first(Z) to follow(Y)

	nullable	first	follow
Z	no	a,b,d	
Y	yes	c	a,b,d
X	no	a,b	

Computing Follow

$Z \rightarrow X\ Y\ Z$ $\quad$ $Y \rightarrow c$ $\quad$ $X \rightarrow a$

$Z \rightarrow d$ $\quad$ $Y \rightarrow \varepsilon$ $\quad$ $X \rightarrow b\ Y\ e$

A. From $X \rightarrow b\ Y\ e$, add {e} to follow(Y)

	nullable	first	follow
Z	no	a,b,d	
Y	yes	c	a,b,d,e
X	no	a,b	

Computing Follow

$Z \rightarrow X\ Y\ Z$ $\quad$ $Y \rightarrow c$ $\quad$ $X \rightarrow a$

$Z \rightarrow d$ $\quad$ $Y \rightarrow \varepsilon$ $\quad$ $X \rightarrow b\ Y\ e$

A. From $Z \rightarrow X\ Y\ Z$ and first(Y) = {c}, add {c} to follow(X)

	nullable	first	follow
Z	no	a,b,d	
Y	yes	c	a,b,d,e
X	no	a,b	c

Computing Follow

$Z \rightarrow X\ Y\ Z$ $\quad$ $Y \rightarrow c$ $\quad$ $X \rightarrow a$

$Z \rightarrow d$ $\quad$ $Y \rightarrow \varepsilon$ $\quad$ $X \rightarrow b\ Y\ e$

B. From $Z \rightarrow X\ Y\ Z$, Y nullable,
add first(Z) = {a,b,d} to follow(X)

	nullable	first	follow
Z	no	a,b,d	
Y	yes	c	a,b,d,e
X	no	a,b	a,b,c,d

Computing Follow

$Z \rightarrow X\ Y\ Z$ $\quad$ $Y \rightarrow c$ $\quad$ $X \rightarrow a$

$Z \rightarrow d$ $\quad$ $Y \rightarrow \varepsilon$ $\quad$ $X \rightarrow b\ Y\ e$

Closure: There are no further additions to follow.

	nullable	first	follow
Z	no	a,b,d	{ }
Y	yes	c	a,b,d,e
X	no	a,b	a,b,c,d

Constructing the LL(1) Parse Table

- The table has non-terminals as row headers, terminals as column headers.
- The entries are productions to be applied for the corresponding combination stack-top and input symbol.
- Entries should be unique, or empty (reject). If two productions are placed in one entry, the grammar is not LL(1).

Constructing the LL(1) Parse Table

For each production X → γ

A. If $\sigma \in$ First(γ)

then enter X → γ in row X, col σ.

[X can produce σ if X → γ used.]

B. If γ is nullable and $\sigma \in$ Follow(X)

enter X → γ in row X, col σ.

[something below X can produce σ.]

Grammar:

$Z \rightarrow X\ Y\ Z$
$Z \rightarrow d$
$Y \rightarrow c$
$Y \rightarrow \varepsilon$
$X \rightarrow a$
$X \rightarrow b\ Y\ e$

Computed Sets:

	nullable	first	follow
Z	no	a,b,d	{ }
Y	yes	c	a,b,d,e
X	no	a,b	a,b,c,d

Parse Table:

A. If $\sigma \in$ First(γ)
then enter $X \rightarrow \gamma$ in row X, col σ.

$Z \rightarrow X\ Y\ Z$ and First(X) = {a, b}

	a	b	c	d	e
Z	$Z \rightarrow X\ Y\ Z$	$Z \rightarrow X\ Y\ Z$			
Y					
X					

Grammar:

Z → X Y Z
Z → d

Y → c
Y → ε

X → a
X → b Y e

Computed Sets:

	nullable	first	follow
Z	no	a,b,d	{ }
Y	yes	c	a,b,d,e
X	no	a,b	a,b,c,d

Parse Table:

A. If σ ∈ First(γ)
then enter X → γ in row X, col σ.

Z → d and First(X) = {d}

	a	b	c	d	e
Z	Z → X Y Z	Z → X Y Z		**Z → d**	
Y					
X					

Grammar:

$Z \rightarrow X\ Y\ Z$

$Z \rightarrow d$

$Y \rightarrow c$

$Y \rightarrow \varepsilon$

$X \rightarrow a$

$X \rightarrow b\ Y\ e$

Computed Sets:

	nullable	first	follow
Z	no	a,b,d	{ }
Y	yes	c	a,b,d,e
X	no	a,b	a,b,c,d

Parse Table:

A. If $\sigma \in$ First(γ)
then enter $X \rightarrow \gamma$ in row X, col σ.

$X \rightarrow a$ and First(a) = {a}

	a	b	c	d	e
Z	$Z \rightarrow X\ Y\ Z$	$Z \rightarrow X\ Y\ Z$		$Z \rightarrow d$	
Y					
X	**$X \rightarrow a$**				

Grammar:

Z → X Y Z
Z → d

Y → c
Y → ε

X → a
X → b Y e

Computed Sets:

	nullable	first	follow
Z	no	a,b,d	{ }
Y	yes	c	a,b,d,e
X	no	a,b	a,b,c,d

Parse Table:

A. If σ ∈ First(γ)
then enter X → γ in row X, col σ.

X → b Y e and First(b Y e) = {b}

	a	b	c	d	e
Z	Z → X Y Z	Z → X Y Z		Z → d	
Y					
X	X → a	**X → b Y e**			

Grammar:

Z → X Y Z
Z → d

Y → c
Y → ε

X → a
X → b Y e

Computed Sets:

	nullable	first	follow
Z	no	a,b,d	{ }
Y	yes	c	a,b,d,e
X	no	a,b	a,b,c,d

Parse Table:

A. If σ ∈ First(γ)
then enter X → γ in row X, col σ.

Y → c and First(c) = {c}

	a	b	c	d	e
Z	Z → X Y Z	Z → X Y Z		Z → d	
Y			**Y → c**		
X	X → a	X → b Y e			

Grammar:

$Z \rightarrow X\ Y\ Z$

$Z \rightarrow d$

$Y \rightarrow c$

$Y \rightarrow \varepsilon$

$X \rightarrow a$

$X \rightarrow b\ Y\ e$

Computed Sets:

	nullable	first	follow
Z	no	a,b,d	{ }
Y	yes	c	a,b,d,e
X	no	a,b	a,b,c,d

Parse Table:

B. If γ is nullable and $\sigma \in$ Follow(X) enter $X \rightarrow \gamma$ in row X, col σ.

follow(Y) = {a,b,d,e},

Done
Empty entries designate reject.

	a	b	c	d	e
Z	$Z \rightarrow X\ Y\ Z$	$Z \rightarrow X\ Y\ Z$		$Z \rightarrow d$	
Y	**$Y \rightarrow \varepsilon$**	**$Y \rightarrow \varepsilon$**	$Y \rightarrow c$	**$Y \rightarrow \varepsilon$**	**$Y \rightarrow \varepsilon$**
X	$X \rightarrow a$	$X \rightarrow b\ Y\ e$			

Example LL(1) Parse using Table

	a	b	c	d	e
Z	Z → X Y Z	Z → X Y Z		Z → d	
Y	**Y → ε**	**Y → ε**	Y → c	**Y → ε**	**Y → ε**
X	X → a	X → b Y e			

Remaining Input	Stack	Use Production
b e c a d	Z	Z → X Y Z
b e c a d	X Y Z	X → b Y e
e c a d	Y e Y Z	Y → ε
e c a d	e Y Z	match
c a d	Y Z	Y → c
a d	Z	Z → X Y Z
a d	X Y Z	X → a
d	Y Z	Y → ε
d	Z	Z → d
ε	ε	
	Accept	

CYK Algorithm

- CYK = “Cocke, Younger, Kasami”, independent discoverers of the same algorithm.
- Deterministic, **not using pda model**.
- Does not require much transformation of the grammar, only to CNF.

CYK Motivation

- Problem is to determine whether a given string x is in L(G).
- A **naïve** approach, would be to generate all possible strings, in increasing length, until either:
  - x is generated, **or**
  - alll strings of length x or shorter have been generated.
- This is a very **slow** process; could be exponential time in the length of the string (e.g. if each symbol could have been generated by two different productions: 2x2x…x2 n times).

CYK Motivation

- Rather than working from the start symbol toward generated strings, might be better to work from string backward, to see if start symbol could have generated the string.
- This too could be exponential if not done carefully.
- The CYK uses “dynamic programming” to make the process efficient.

Dynamic Programming?

- Recursive expressions, such as
  f(n) = f(n-1) + f(n-3); f(n) = 1 if n < 3;
  while **very clear** in their intent, may be **inefficient** if taken literally.
- They may entail much **recomputation** unless care is taken to ensure otherwise.
- The idea of dynamic programming is to compute from the "**bottom up**":
  f(0), f(1), f(2), f(3), …
  remembering values for **posterity** even if they might not be used ultimately.

CYK Algorithm

- Let $x = x_1 x_2 \ldots x_n$ be the string to be parsed.
- Define a(i, j) =

$$\{B \mid B \Rightarrow^* x_i x_{i+1} \ldots x_j\}$$

for each $i, j \in \{1, 2, \ldots, n\}$ with $i \leq j$.

- So $x \in L(G)$ iff $S \in a(1, n)$.
- The essence of CYK is how to compute a(1, n) efficiently.

CYK Algorithm

- Assume that the grammar is Chomsky Normal Form.
- We arrive at a(1, n) by the following method:
  - First compute a(i, i) for each i:

    $a(i, i) = \{B \mid B \rightarrow x_i\}$

- Since the grammar is in CNF, the indicated productions are the only ones that produce terminal symbols.

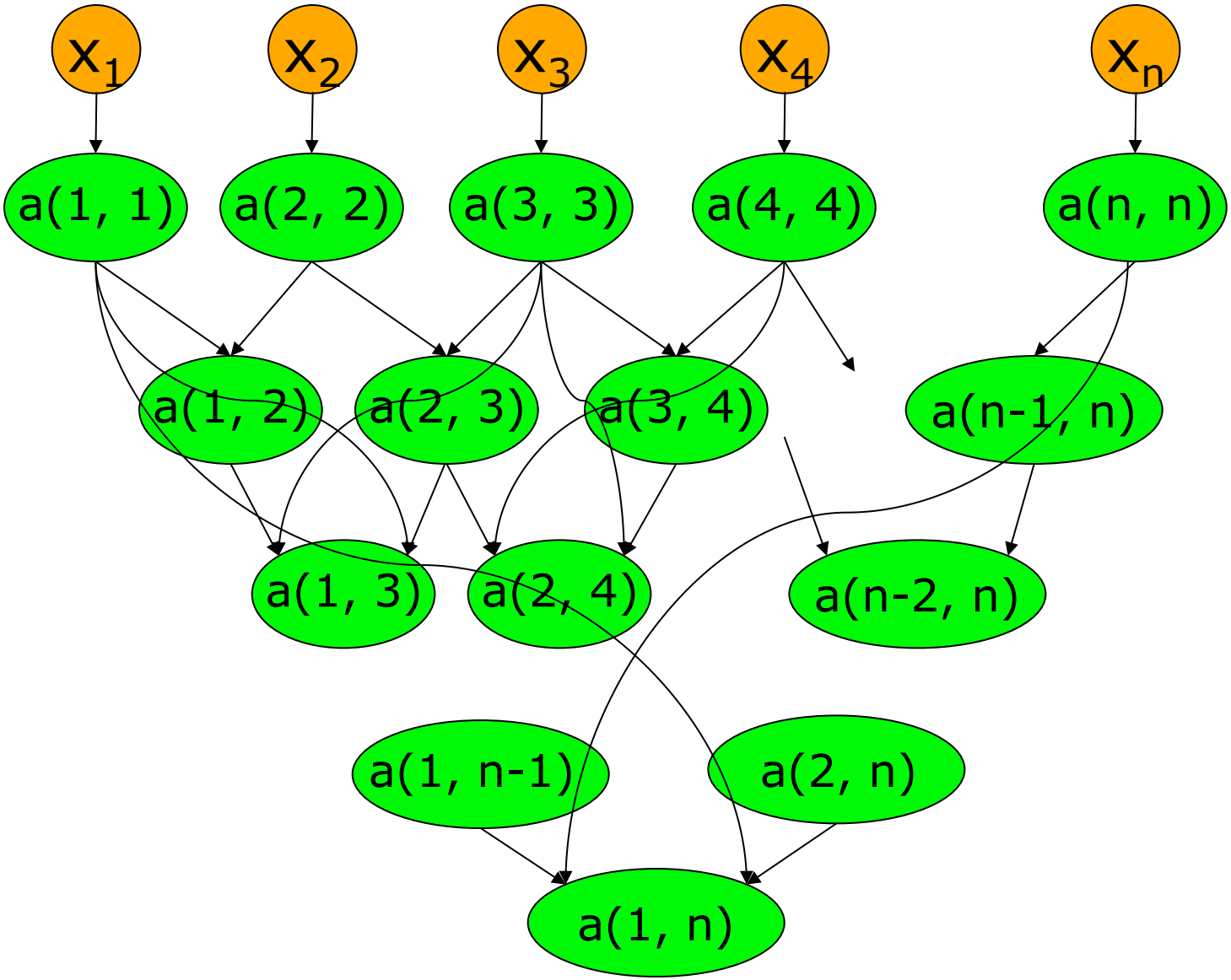
CYK Algorithm

- Next compute a(i, i+1) for each i, then a(i, i+2) for each i, and so on.
- To compute a(i, j) in general, we use pairs, which have been computed in the previous iteration:

  a(i, i) a(i+1, j)
  a(i, i+1) a(i+2, j)
  a(i, i+2) a(i+3, j)
  …
  a(i, j-1) a(j, j)

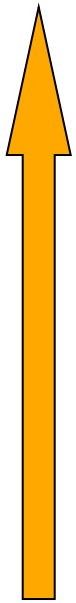
- For each pair, we need only to check whether there is a production of the form $A \rightarrow BC$ where $B \in a(i, i+k)$ and $C \in a(i+k-1, j)$ since the grammar is in CNF.

CYK Algorithm Data Flow



CYK “Wavefront” Matrix

a(1, 1)	a(1, 2)	a(1, 3)	a(1, 4)		a(1, n-1)	a(1, n)
	a(2, 2)	a(2, 3)	a(2, 4)		a(2, n-1)	a(2, n)
		a(3, 3)	a(3, 4)		a(3, n-1)	a(3, n)
			a(4, 4)			
					a(n-1, n-1)	a(n-1, n)
						a(n, n)



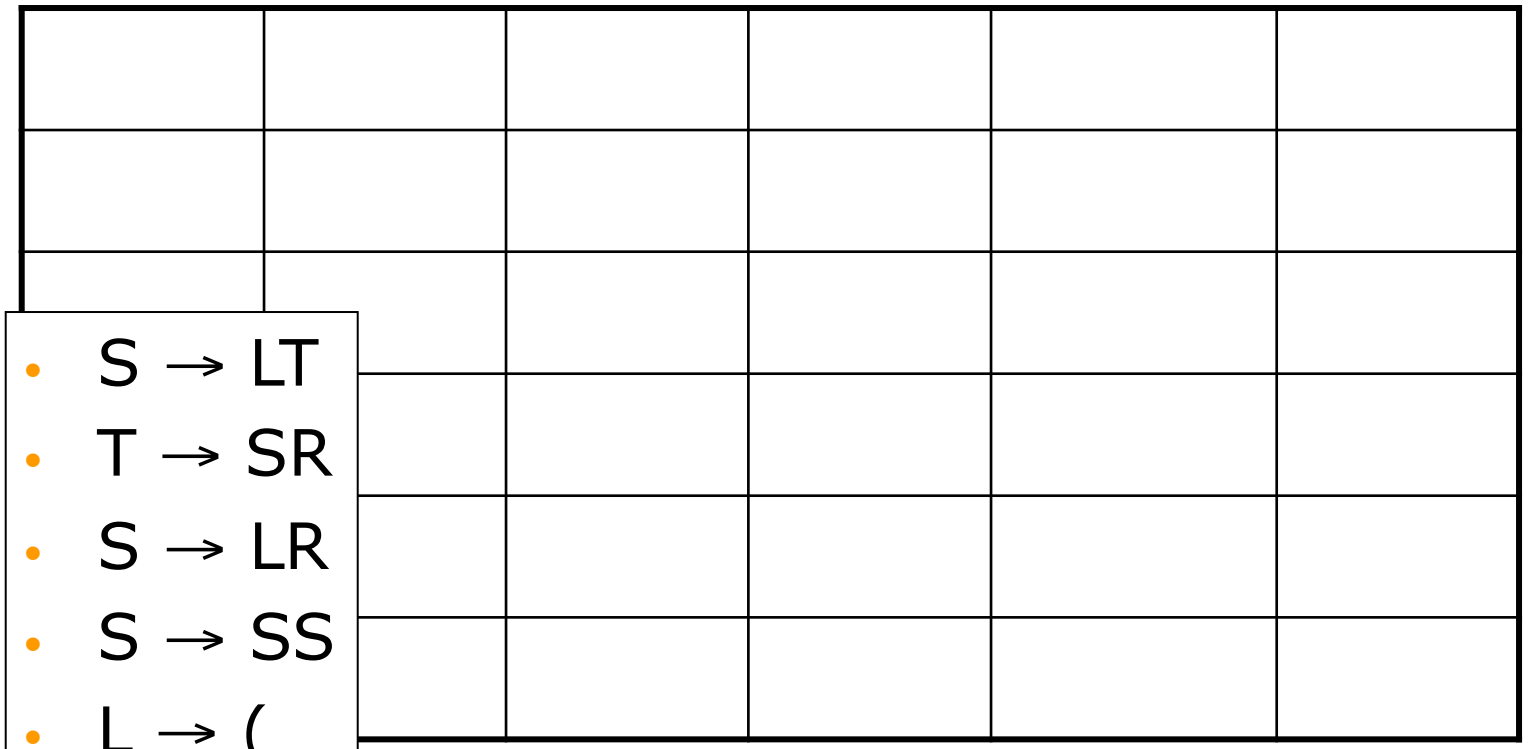
Each entry is computed from entries in its same row and column, e.g.
a(1, 4) from
a(1,1) and a(2, 4),
a(1, 2) and a(3, 4),
a(1, 3) and a(4, 4).

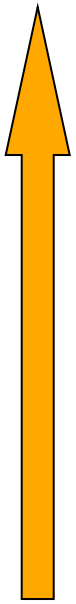
CYK Example

- S → LT
- T → SR
- S → LR
- S → SS
- L → (
- R →)

- This grammar is in CNF.

CYK Movie on input (()())

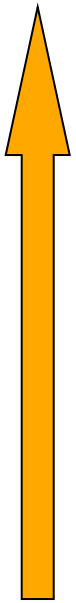




- S → LT
- T → SR
- S → LR
- S → SS
- L → (
- R →)

CYK Movie on input (()())

L					
	L				
		R			
			L		
				R	
					R



- S → LT
- T → SR
- S → LR
- S → SS
- L → (
- R →)

CYK Movie on input (()())

L	∅				
	L	S			
		R	∅		
			L	S	
				R	∅
					R

- S → LT
- T → SR
- S → LR
- S → SS
- L → (
- R →)

CYK Movie on input (()())

L	∅	∅			
	L	S	∅		
		R	∅	∅	
			L	S	T
				R	∅
					R

- S → LT
- T → SR
- S → LR
- S → SS
- L → (
- R →)

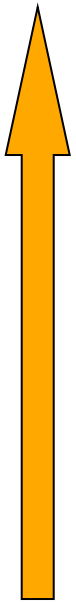
CYK Movie on input (()())

L	∅	∅	∅		
	L	S	∅	S	
		R	∅	∅	∅
			L	S	T
				R	∅
					R

- S → LT
- T → SR
- S → LR
- S → SS
- L → (
- R →)

CYK Movie on input (()())

L	∅	∅	∅	∅	
	L	S	∅	S	T
		R	∅	∅	∅
			L	S	T
				R	∅
					R



- S → LT
- T → SR
- S → LR
- S → SS
- L → (
- R →)

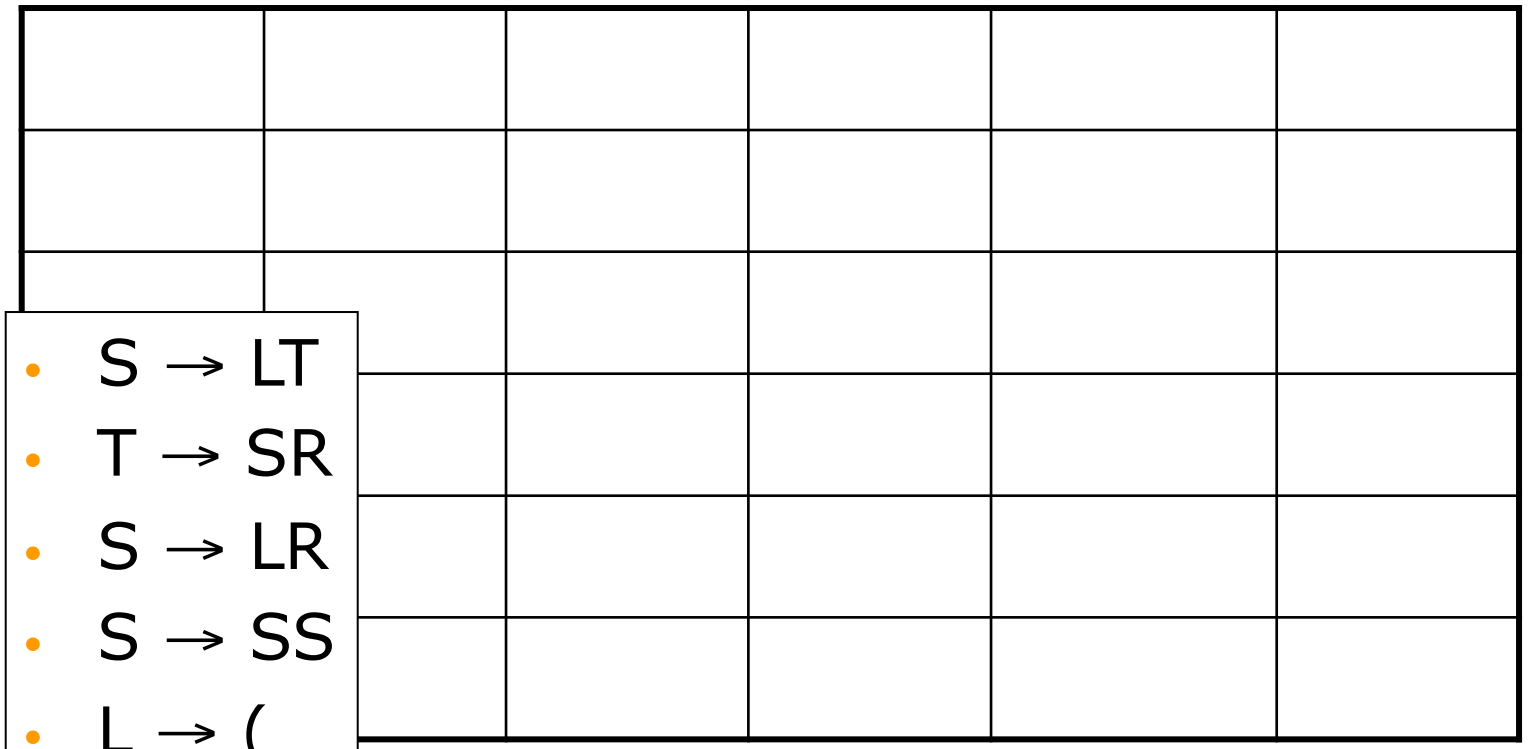
CYK Movie on input (()())

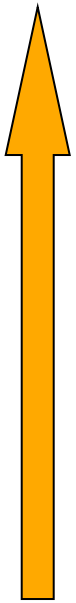
The string (()()) is accepted.

L	∅	∅	∅	∅	S
	L	S	∅	S	T
		R	∅	∅	∅
			L	S	T
				R	∅
					R

- S → LT
- T → SR
- S → LR
- S → SS
- L → (
- R →)

CYK Movie on input ())())





- S → LT
- T → SR
- S → LR
- S → SS
- L → (
- R →)

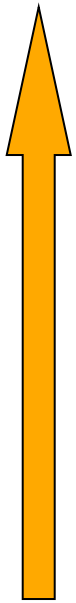
CYK Movie on input ())())

L	S				
	R	∅			
		R	∅		
			L	S	
				R	∅
					R

- S → LT
- T → SR
- S → LR
- S → SS
- L → (
- R →)

CYK Movie on input ())())

L	S	T			
	R	∅	∅		
		R	∅	∅	
			L	S	T
				R	∅
					R



- S → LT
- T → SR
- S → LR
- S → SS
- L → (
- R →)

CYK Movie on input ())())

L	S	T	∅		
	R	∅	∅	∅	
		R	∅	∅	∅
			L	S	T
				R	∅
					R

- S → LT
- T → SR
- S → LR
- S → SS
- L → (
- R →)

CYK Movie on input ())())

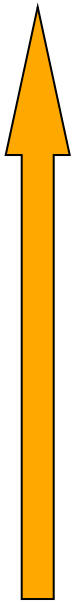
L	S	T	∅	∅	
	R	∅	∅	∅	∅
		R	∅	∅	∅
			L	S	T
				R	∅
					R

- S → LT
- T → SR
- S → LR
- S → SS
- L → (
- R →)

CYK Movie on input ())())

The string ())()) is not accepted.

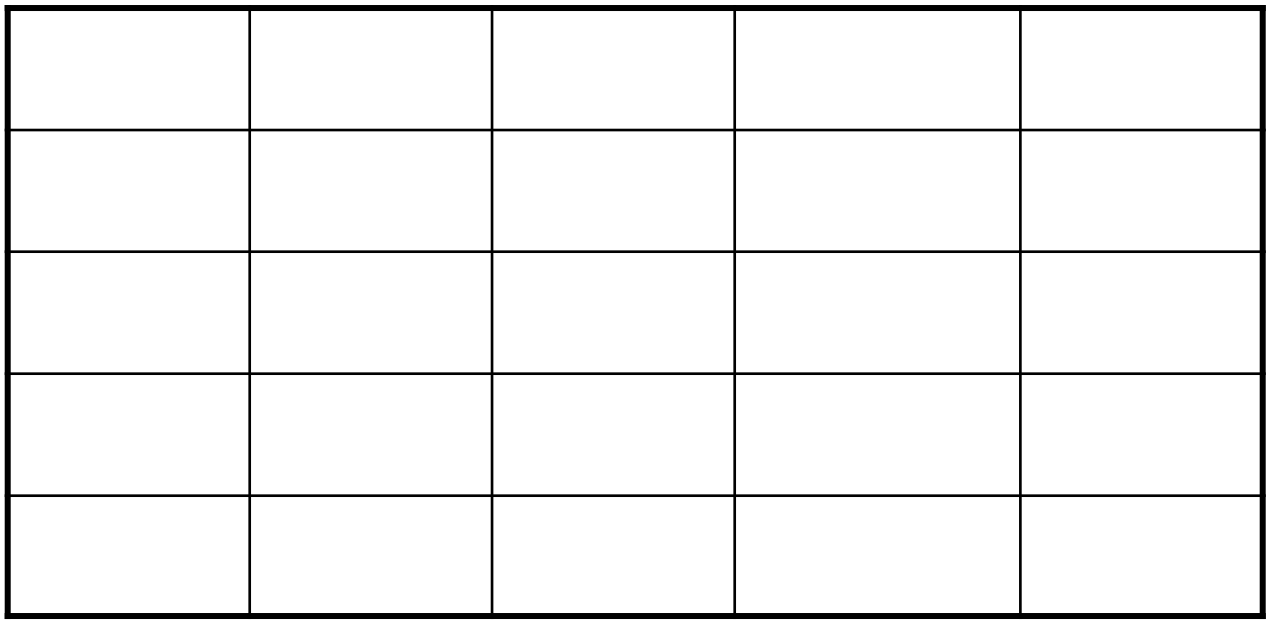
L	S	T	∅	∅	∅
	R	∅	∅	∅	∅
		R	∅	∅	∅
			L	S	T
				R	∅
					R

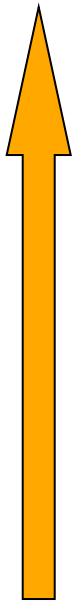


- S → LT
- T → SR
- S → LR
- S → SS
- L → (
- R →)

CYK Movie on input abbaa (different grammar)

- S → AB
- S → BC
- A → AB
- A → a
- B → AA
- B → CB
- B → b
- C → a
- C → b

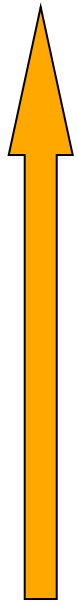




CYK Movie on input abbaa (different grammar)

- S → AB
- S → BC
- A → AB
- A → a
- B → AA
- B → CB
- B → b
- C → a
- C → b

A, C				
	B, C			
		B, C		
			A, C	
				A, C



CYK Movie on input abbaa (different grammar)

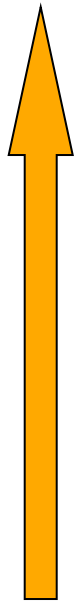
- S → AB
- S → BC
- A → AB
- A → a
- B → AA
- B → CB
- B → b
- C → a
- C → b

A, C	S, A, B			
	B, C	S, B		
		B, C	S	
			A, C	B
				A, C

CYK Movie on input abbaa (different grammar)

- S → AB
- S → BC
- A → AB
- A → a
- B → AA
- B → CB
- B → b
- C → a
- C → b

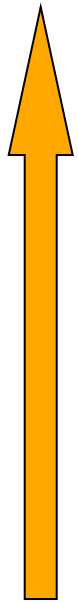
A, C	S, A, B	S, A, B		
	B, C	S, B	S	
		B, C	S	B
			A, C	B
				A, C



CYK Movie on input abbaa (different grammar)

- S → AB
- S → BC
- A → AB
- A → a
- B → AA
- B → CB
- B → b
- C → a
- C → b

A, C	S, A, B	S, A, B	S, B	
	B, C	S, B	S	B
		B, C	S	B
			A, C	B
				A, C



CYK Movie on input abbaa (different grammar)

The string abbaa is accepted.

- S → AB
- S → BC
- A → AB
- A → a
- B → AA
- B → CB
- B → b
- C → a
- C → b

A, C	S, A, B	S, A, B	S, B	S, A, B
	B, C	S, B	S	B
		B, C	S	B
			A, C	B
				A, C

Complexity of CYK

- n is the length of the input string.
- p is the number of productions, which can be treated as a constant.
- There are $O(n^2)$ sets to be computed.
- Each set can be represented as a **bit-vector**, so that elements can be added and membership-tested in O(1) time.
- A general set can be computed in O(n) iterations, each iteration involving examination of p productions.
- The total time is proportional to $pn\ n^2 \in O(n^3)$.

Expressing CYK Iteratively

- Work by super-diagonals d = 2, 3, …, n
- d = 2 compute:
  - a(1, 2) = a(1, 1)⊗a(2, 2)
  - a(2, 3) = a(2, 2)⊗a(3, 3)
  - …
  - a(n-1, n) = a(n-1, n-1)⊗a(n, n)
- d = 3 compute:
  - a(1, 3) = a(1, 1)⊗a(2, 3) ∪ a(1, 2)⊗a(3, 3)
  - a(2, 4) = a(2, 2)⊗a(3, 4) ∪ a(2, 3)⊗a(4, 4)
  - …
  - a(n-2, n) = a(n-2, n-2)⊗a(n-1, n) ∪ a(n-2, n-1)⊗a(n, n)
- . . .
- d = n compute:
  - a(1, n) = a(1, 1)⊗a(2, n) ∪ a(1, 2)⊗a(3, n) ∪ . . .
    a(1, n-1)⊗a(n-1, n)

⊗ is pairing

Summary of CYK

- Grammar is in CNF, input is $x = x_1 x_2 \ldots x_n$.
- **For** r = 1 to n // diagonal
  a(r, r) = {B | B → x_r}
- **For** d = 2 to n // super-diagonals
  For r = 1 to n-d+1 // row
  c = r+d-1; // column
  a(r, c) = ∅; // entry to compute
  For k = r to c-1
  For each production B → CD
  If C ∈ a(r, k)
  and D ∈ a(k+1, c)
  add B to a(r, c);
- x is in the language iff S ∈ a(n, n).

Additional Things to Ponder

- Simplify indexing expressions, e.g. by using transpose of matrix
- Recover derivation tree(s) from the algorithm

Real-World Parsing

- The value of CYK is an upper-bound for **arbitrary** context-free languages.
- $O(n^3)$ is too slow for programming languages, where O(n) is desired (and doable for suitably-deterministic languages).
- May be acceptable for natural languages or other special languages.

Non-PL Applications of Parsing

- Natural Language
- Genomics (RNA strings)
  - Stochastic Context-Free Grammars
  - Transformational Grammars (transform one string to another)
- Music
  - Explaining a passage of music
- Art
  - Understanding a piece of art

CYK Application: Impro-Visor Roadmaps

- Impro-Visor offers a feature of producing "roadmaps" which explain harmonic progressions in terms of idiomatic chunks (called "bricks")

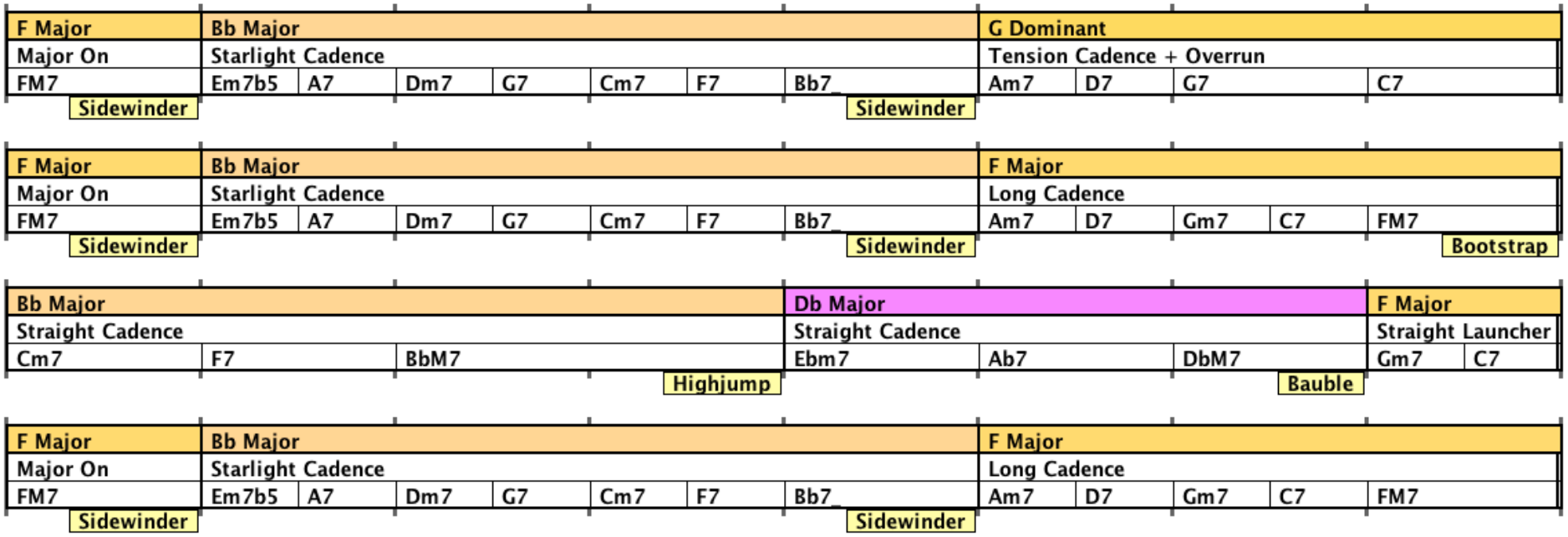
- The input is a string representing the chord progression of a tune, such as:

FM7 | Em7b5 A7 | Dm7 G7 | Cm7 F7 | Bb7_ | Am7 D7 | G7 | C7 |
FM7 | Em7b5 A7 | Dm7 G7 | Cm7 F7 | Bb7_ | Am7 D7 | Gm7 C7 | FM7 |
Cm7 | F7 | BbM7 | / | Ebm7 | Ab7 | DbM7 | Gm7 C7 |
FM7 | Em7b5 A7 | Dm7 G7 | Cm7 F7 | Bb7_ | Am7 D7 | Gm7 C7 | FM7 |

Impro-Visor Roadmap

The output is a roadmap for the tune:

Confirmation



Use of CYK Parsing

- The idiomatic bricks that can be used in a roadmap are defined by an ambiguous grammar.
- The CYK algorithm parses the input reduces chord subsequences to sets of bricks.
- An additional cost-minimization pass produces the “best” roadmap for the tune.
- This is work with Xanda Schofield, August Toman-Yih, Zach Merritt, and John Elliot. It will be published in the Computer Music Journal.

Aside: L-Systems

- With grammars, productions are applied sequentially at any site in the sentential form where the LHS matches.

- With L-systems, productions are applied at all sites simultaneously, sort of like parallel processing or cellular automata.

Example: Fibonacci L-System

- Production
  - a → b
  - b → ba
- Derivation

  a
  b
  ba
  bab
  babba
  babbabab
  babbababbabba
  . . .

Example: Thue Sequence L-System

- Production
  - a → ab
  - b → ba
- Derivation

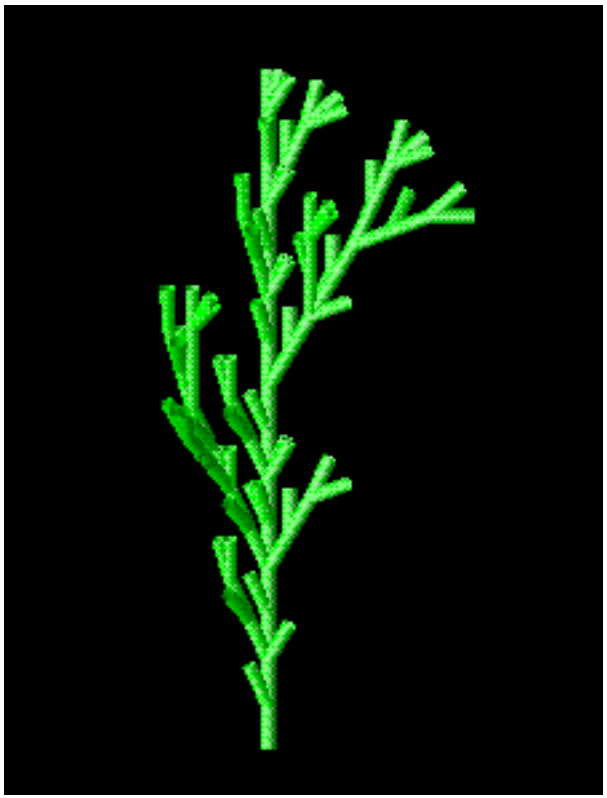
  ab
  abba
  abbabaab
  abbabaabbaababba
  abbabaabbaababbabaababbaabbabaab
  . . .

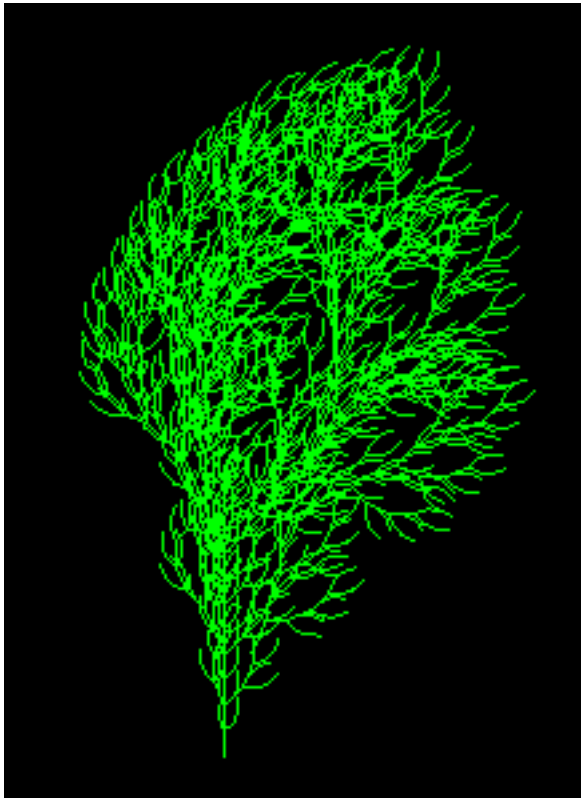
- Distinctions
  - Self-similar (fractal) nature.
  - There is a limit infinite sequence containing all strings as prefixes.
  - No string has a subsequence of the form xxx.

Applications of L-Systems?

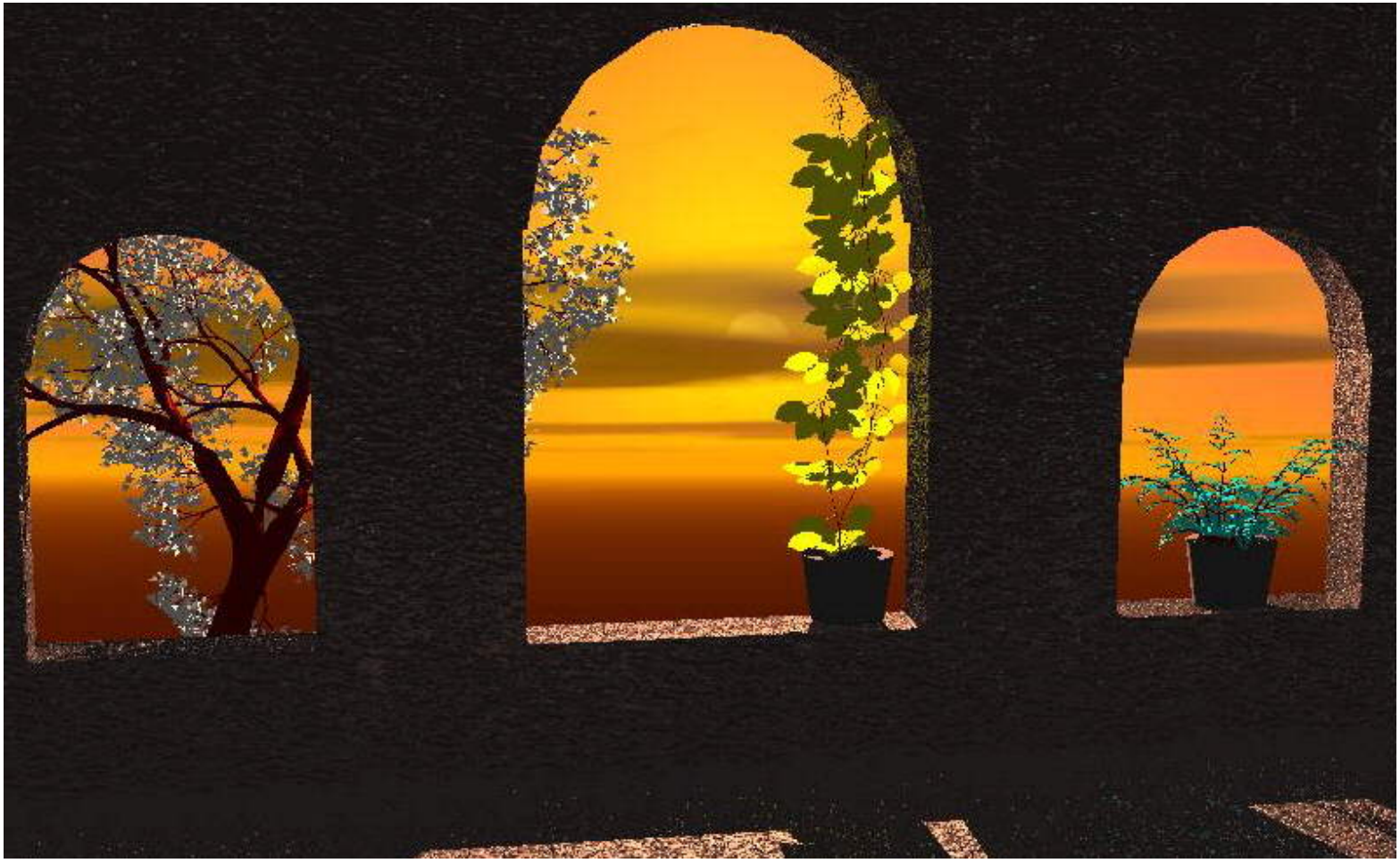
- Studied extensively as models for biological development (particularly plants)
- Letters in L sequence can be interpreted as commands to add features, something like “turtle graphics”

Plants constructed using L-Systems





L-System Garden



More Examples:

- Przemyslaw Prusinkiewicz, The Algorithmic Beauty of Plants, Springer-Verlag, 1990.