SCC.312
Languages and Compilation
(Week 13)

Paul Rayson

p.rayson@lancaster.ac.uk

Last week's topics

Reminder

- Regular Grammars
- Language of the grammar – L(G)
- Finite State Recognisers
  - Non-deterministic and deterministic
  - Subset construction algorithm
- Regular Expressions
- Equivalence

Chomsky Hierarchy

Type	Grammar	Machine	Other Equivalent
3	Regular	Finite State Recogniser	Regular Expressions

This week’s Lectures

- The “Repeat State” Theorem

A COMPREHENSIVE GRAMMAR OF THE ENGLISH LANGUAGE

Randolph Quirk
Sidney Greenbaum
Geoffrey Leech
Jan Svartvik

LONGMAN

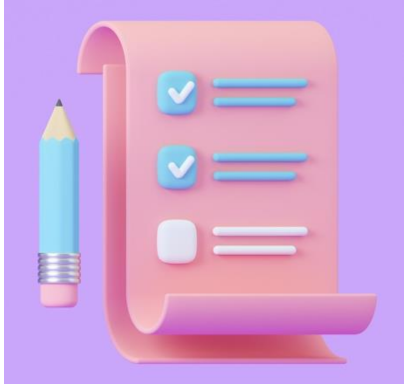
Context free Grammars

Pushdown Recognisers

- Uses of Context Free Grammars

- Syntax, Semantics and Ambiguity

Learning objectives



1. identify the format of rules in context free grammars
2. understand the limitations of context free grammars in representing strings
3. understand the concept of equivalence between two grammars
4. understand the concept of ambiguity in a grammar
5. generate a pushdown recogniser (PDR) that corresponds to a given context free grammar
6. understand the difference between deterministic and a non-deterministic context free grammar

Already covered last week

Regular Grammars

Reminder

- All productions are one of two formats:
  - **X → yZ** (X, Z non terminals, y is a terminal)
  - **X → t** (Note: t can be the empty string)
- Converted into Finite State Recognisers
- Recognise string of length *n* in *n* steps

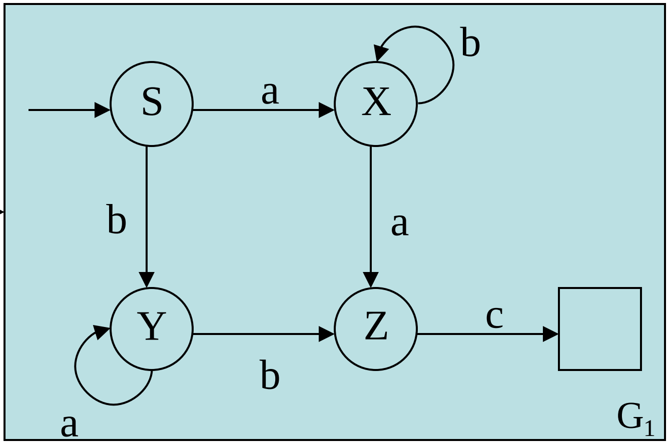
$S \rightarrow aX \mid bY$

$X \rightarrow bX \mid aZ$

$Y \rightarrow aY \mid bZ$

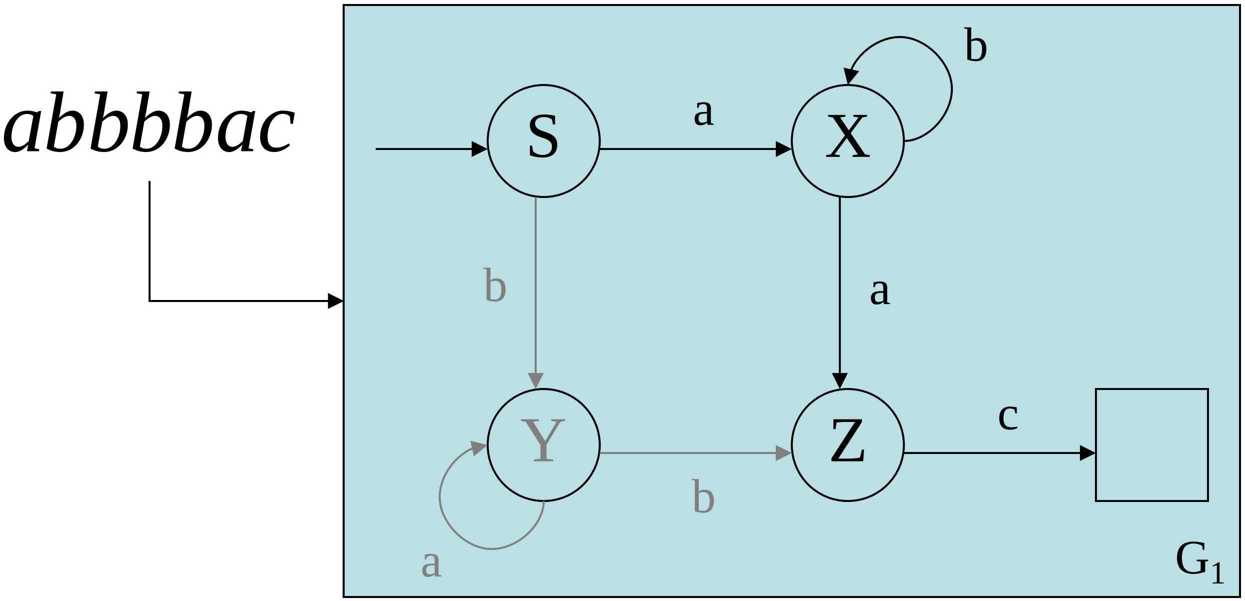
$Z \rightarrow c$

G_1



Using the Finite State Recogniser

- If a FSR can accept strings of infinite length then it must contain a loop
  - – passes through a state more than once



“Repeat State” Theorem – “*vwx*”

- The string that is accepted can be expressed as 3 concatenated sub strings – *vwx*
  - *v* is the string accepted before the repeat state(s)
  - *w* is the repeated string
  - *x* is the string accepted after the repeat state(s)
    - *v* and *x* could be empty, but *w* is non-empty
- vw^ix for all $i \geq 0$
- Applies to any infinite regular language

Example

- "Applies to any infinite regular language"
- vw^ix for all i ≥ 0
- e.g. ab^iac for G_1
    - $v = a$
    - $w^i = b^i$
    - $x = ac$

Using the Repeat State Theorem

- We can use the theorem to determine if a language is regular
- For example $\{a^i b^i : i \geq 1\}$ looks like it might be regular
- The *Repeat State Theorem* says we need a “repeatable sub string” in vw^ix
  - What is it? *ab*? *a*? *b*?

FSRs and Counting

- Also if $\{a^i b^i : i \geq 1\}$ was regular we could parse it with a FSR
  - Needs to count the number of ‘*a*’s and check that there were the same number of ‘*b*’s
- FSRs can only count by going into a new state
  - As our FSR only has M states it cannot count above M
- This counted pattern is infinite so it cannot be regular

Note: some regular languages can be infinite

Beyond Regular Languages

- By using the Repeat State Theorem and the counting limitations of FSRs we have shown that $\{a^i b^i : i \geq 1\}$ is not regular
- A similar argument can be used to check for matching parentheses (i.e. in a compiler)
- Obviously will need something more powerful than a regular grammar

A simple Non-Regular Grammar

- terminals: **a b**
- non-terminals: **S**
- distinguished symbol: **S**
- productions:
    - S → ab
    - S → aSb

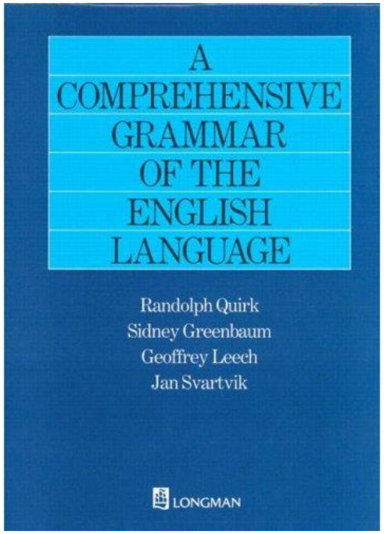
A Derivation from the Grammar

- – S
- – aSb
- – aaSbb
- – aaabbb

- this grammar generates
  $\{ a^i b^i : i \geq 1\}$

Context Free Grammars

Type 2



Context Free Grammars

- The next type of phrase structure grammar is context free grammars that generate context free languages (CFLs)
- The productions have the format
  - **X → RHS**
    - X is a single non-terminal,
    - RHS (right hand side) is any mixture of terminals and/or non-terminals, and can be empty
- Regular grammars are a restricted form of context free grammars

Identifying a CF Grammar

Is this a context free grammar?

$$S \rightarrow aSYZ \mid aYZ$$

$$ZY \rightarrow YZ$$

$$aY \rightarrow ab$$

$$bY \rightarrow bb$$

$$bZ \rightarrow bc$$

$$cZ \rightarrow cc$$

Not a CF Grammar

Identifying a CF Grammar

Are either of these two grammars CF?

$S \rightarrow aB \mid bA \mid \varepsilon$

$A \rightarrow aS \mid bAA$

$B \rightarrow bS \mid aBB$

A Context Free Grammar

$S \rightarrow aS \mid aB \mid bC$

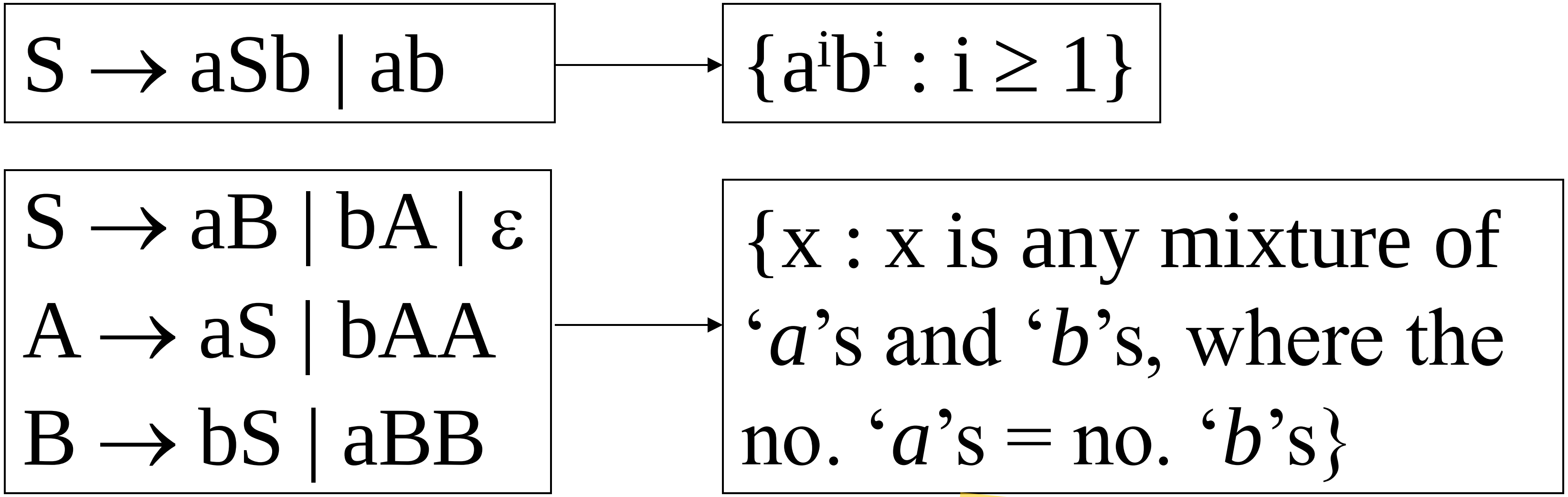
$B \rightarrow bC$

$C \rightarrow cC \mid c$

A Regular Grammar

L(G) for some Examples

- These examples are context free



$S \rightarrow aSb \mid ab$

$\{a^i b^i : i \geq 1\}$

$S \rightarrow aB \mid bA \mid \varepsilon$

$A \rightarrow aS \mid bAA$

$B \rightarrow bS \mid aBB$

{x : x is any mixture of '*a*'s and '*b*'s, where the no. '*a*'s = no. '*b*'s}

Note the difference: second example is any mixture of 'a's and 'b's in any order

An Example Toy English Grammar

Reminder

R1 - a SENTENCE could be a NOUN-PHRASE followed by a VERB-PHRASE

R2 - a NOUN-PHRASE could be an ARTICLE followed by a NOUN

R3 - (alternatively) a NOUN-PHRASE could be an ARTICLE followed by an ADJECTIVE followed by a NOUN

R4 - a VERB-PHRASE could be a VERB followed by a NOUN-PHRASE

R5 - (alternatively) a VERB-PHRASE could be just a VERB

where, for example,

R6 - an ARTICLE could be *a* or *the*

R7 - a NOUN could be *man*, *doctor*, *dalek*, *boy* or *dog*

R8 - an ADJECTIVE could be *big*, *small* or *red*

R9 - a VERB could be *hates, likes* or *bites*

Context-Free Grammars

- if we look at the productions (string1→ string2) in the first example (the English-language one) in the first week, there is something special about them
- in each production there is exactly one non-terminal (and no terminals) on the left hand side
- a grammar with this property is called **context-free**

A Non-Context-Free Grammar

- also in the first week there was part of a grammar to handle subject-verb agreement:
  - *R6:* NOUN VERB → SINGULAR-NOUN S-FORM-OF-VERB
  - *R7*: NOUN VERB → PLURAL-NOUN SIMPLE-FORM-OF-VERB

- these are not context-free rules

Rewritten as a Context-Free Grammar I

- SENTENCE → SINGULAR-NOUN-PHRASE SINGULAR-VERB-PHRASE
- SENTENCE → PLURAL-NOUN-PHRASE PLURAL-VERB-PHRASE
- SINGULAR-NOUN-PHRASE → ARTICLE SINGULAR-NOUN
- SINGULAR-NOUN-PHRASE → ARTICLE ADJECTIVE SINGULAR-NOUN
- PLURAL-NOUN-PHRASE → ARTICLE PLURAL-NOUN
- PLURAL-NOUN-PHRASE → ARTICLE ADJECTIVE PLURAL-NOUN
- SINGULAR-VERB-PHRASE → S-FORM-OF-VERB NOUN-PHRASE
- SINGULAR-VERB-PHRASE → S-FORM-OF-VERB
- PLURAL-VERB-PHRASE → SIMPLE-FORM-OF-VERB NOUN-PHRASE
- PLURAL-VERB-PHRASE → SIMPLE-FORM-OF-VERB
- NOUN-PHRASE → SINGULAR-NOUN-PHRASE
- NOUN-PHRASE → PLURAL-NOUN-PHRASE

Rewritten as a Context-Free Grammar II

- ARTICLE → a
- ARTICLE → the
- SINGULAR-NOUN → man
- SINGULAR-NOUN → boy
- SINGULAR-NOUN → dog
- PLURAL-NOUN → men
- PLURAL-NOUN → boys
- PLURAL-NOUN → dogs
- ADJECTIVE → big
- ADJECTIVE → small
- ADJECTIVE → red
- SIMPLE-FORM-OF-VERB → hate
- SIMPLE-FORM-OF-VERB → like
- S-FORM-OF-VERB → hates
- S-FORM-OF-VERB → likes

Rewritten as a Context-Free Grammar III

- using this grammar you can derive
  - the man hates the boy
  - the men hate the boy
- but not
  - the man hate the boy
  - the men hates the boy

Grammar Equivalence

Reminder

- the two grammars agree on which strings are grammatical and which are not - the grammars are **equivalent**
- the grammars do not give the same structure to the string - they are only **weakly** equivalent

The Agreement Grammar

- this new grammar (although repetitive) is probably the best way to represent the facts anyway - in a more complete grammar you would want to say that *a* is not an appropriate ARTICLE for a PLURAL-NOUN-PHRASE
- more generally, you would want to say that some ARTICLEs are appropriate for SINGULAR-NOUN-PHRASEs (*a*, *this*) and some for PLURAL-NOUN-PHRASEs (*these*), and some for both (*the, some*)

A Context Free Machine?

- We could not design finite state recognisers that are equivalent to context free languages
  - we need a machine that is more powerful
- It is called a **Pushdown Recogniser** (PDR)
- It is may sometimes be called a Pushdown Automaton (PDA)

Pushdown Recogniser

Type 2

Pushdown Recogniser

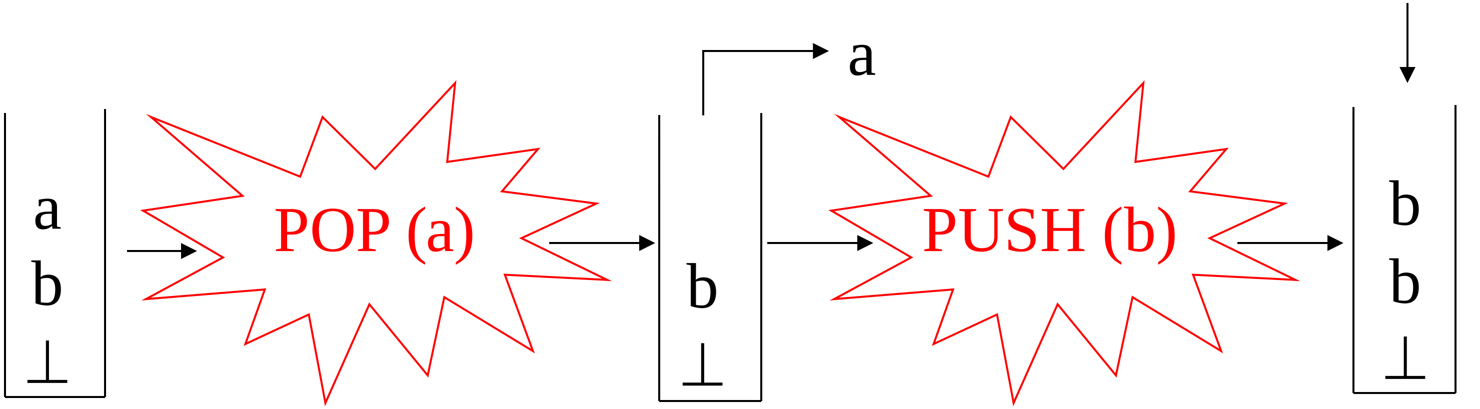
- A pushdown recogniser is a finite state machine with extra storage
- This is called the **stack**
  - It has two procedures, **pop** and **push**
  - It is restricted in accessibility. We can add or remove items only from the top of the stack
  - It is potentially infinite in size
  - It can contain terminals and non-terminals

Remember Stacks from 1st year?

Pushdown Recogniser Stack

- As the PDR reads a symbol, it can remove (**pop)** that symbol off the top of the stack and replace (**push**) it with one or more symbols
- ⊥ is the **stack bottom marker**

JFlap uses Z as stack bottom marker



Pushdown Recogniser Stack

- Can push more than one symbol at a time
  - PUSH(abc) is PUSH(c), PUSH(b), PUSH(a)
  - Easier to allow a sequence
  - Does not affect the power of the PDR
- Pushing the empty string (ε) leaves the stack unchanged
- Not allowed to pop the empty string

JFlap does allow this but in the practicals and coursework you shouldn't do this

Pushdown Recogniser Arcs

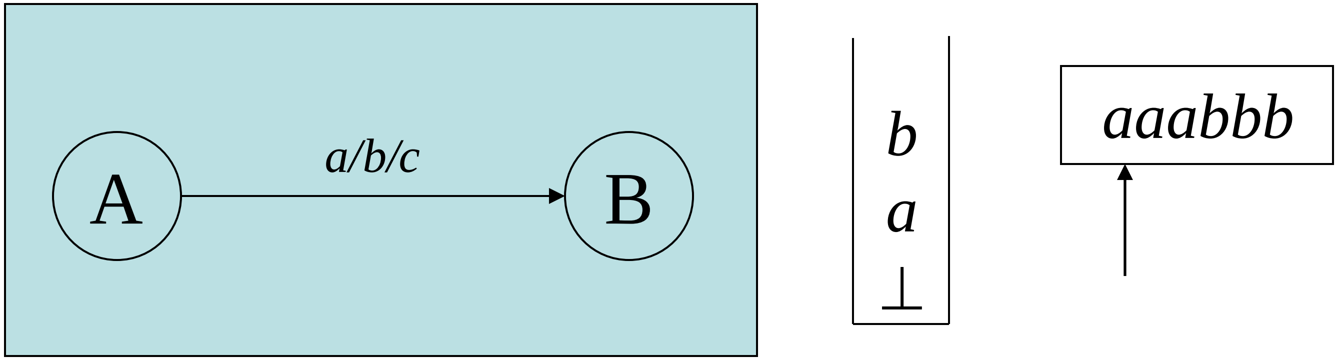
- We show what is popped and what is pushed on the arcs of our push down recogniser using the following format:

  a/b/cd

  - *a/b/cd* – the symbol read in the input string
  - *a/b/cd* – symbol popped off the stack
  - *a/b/cd* – symbol(s) pushed onto the stack

Pushdown Recogniser Arcs

- If in state A, about to read *a*, and *b* is on the top of the stack
- Then POP(*b*), PUSH(*c*) and move to state B



Push Down Recogniser

Stack

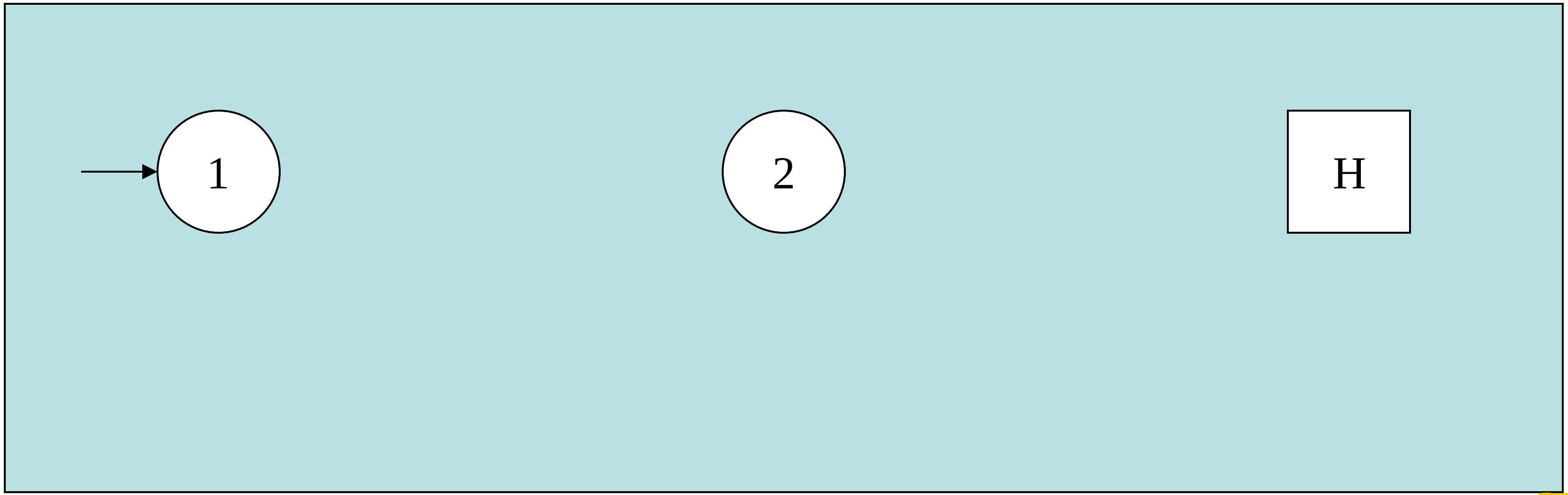
Input String

Pushdown Recogniser Arcs

- The symbol b must be on top of the stack for it to be popped off otherwise an error occurs and we fail to parse the string
- The empty string (ε) can be used to ignore (i.e. not read) the input symbol (e.g. $\varepsilon/b/c$)
  - – Remain positioned at the same input symbol

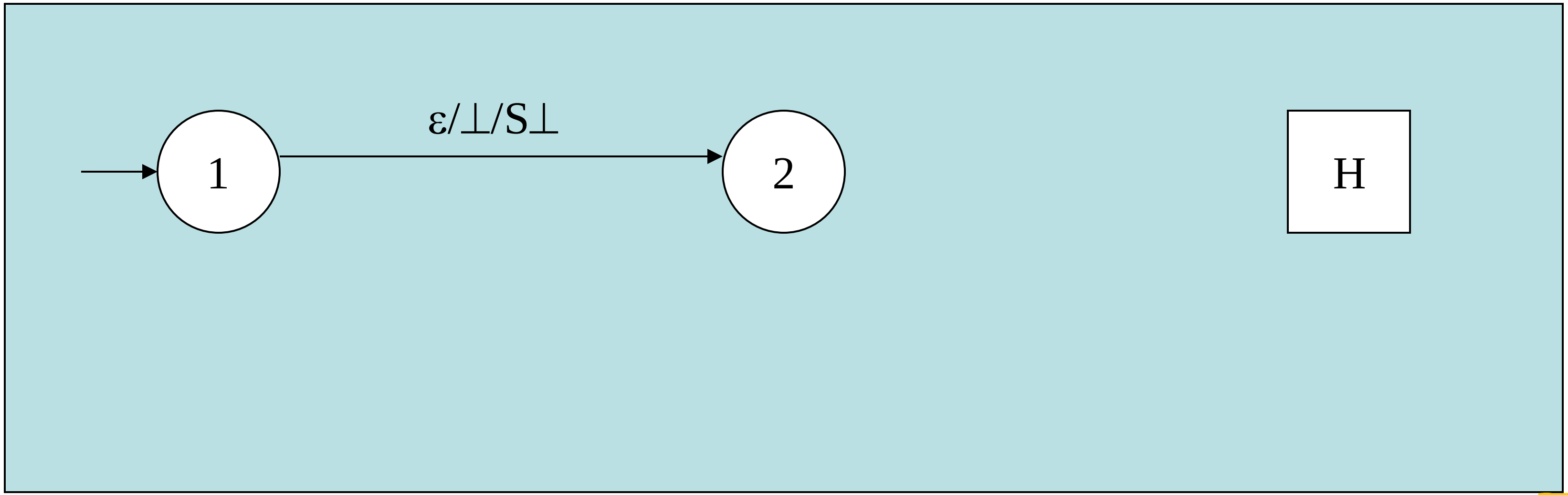
Constructing a PDR

- We mark 3 states – the start state (1), an intermediate state (2) and the halt state (H)
  - Note: in this PDR the states are numbers not letters



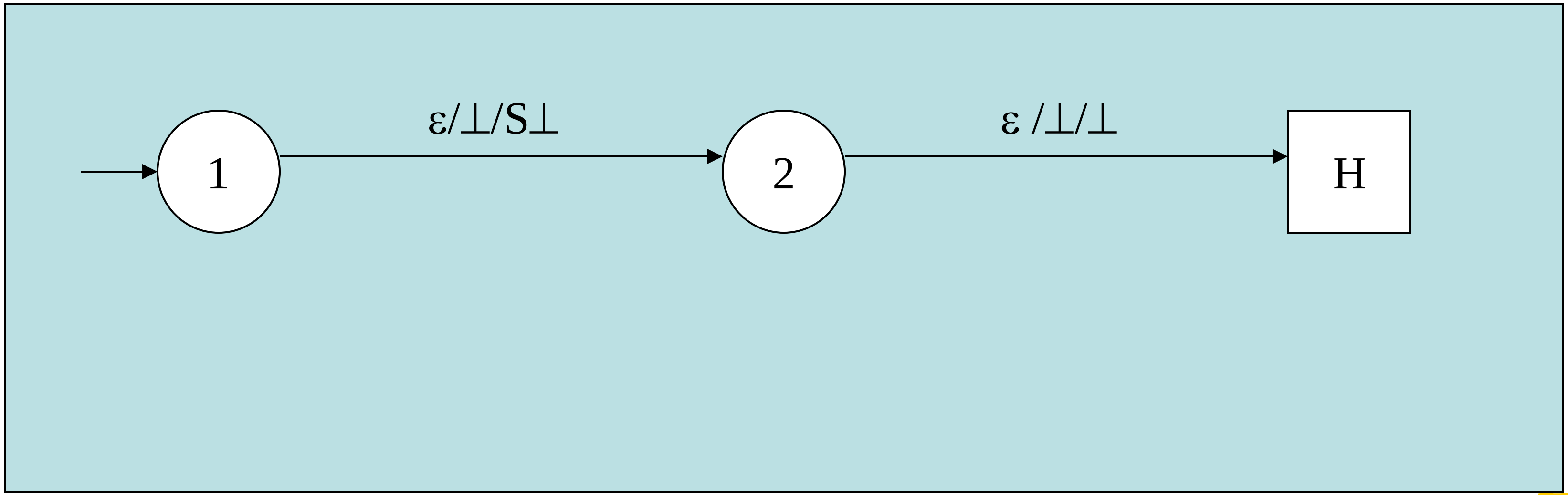
Constructing a PDR

- We add an arc from state 1 to 2 labelled ε/⊥/S⊥
  - – where S is the start symbol



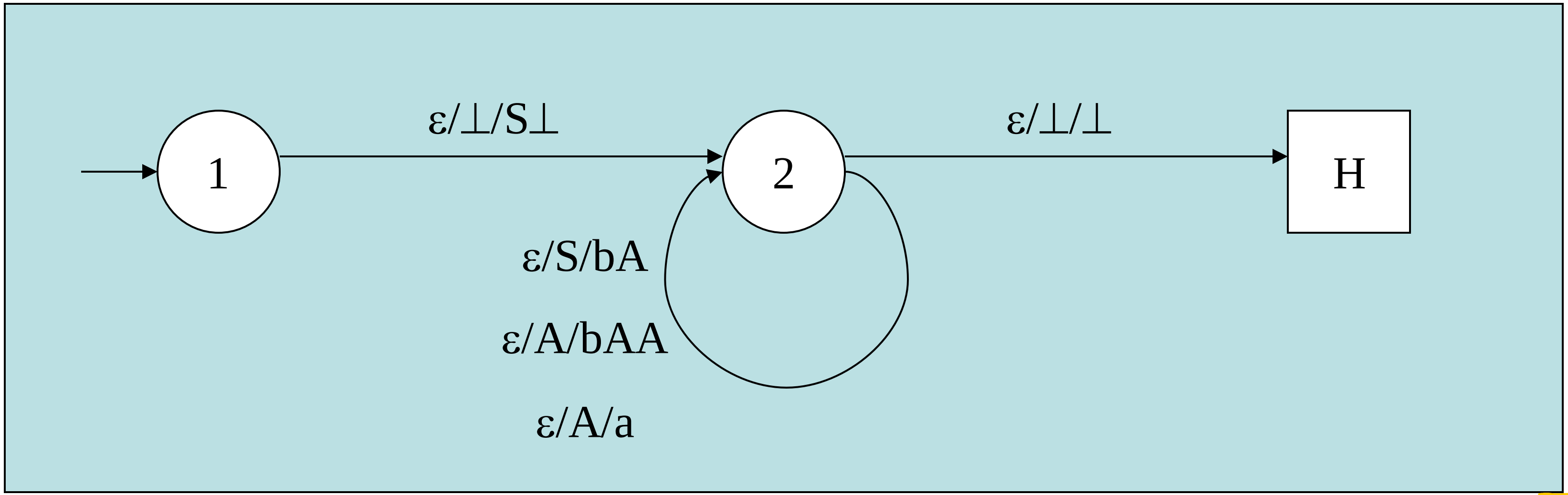
Constructing a PDR

- We add an arc from state 2 to H labelled ε/⊥/⊥



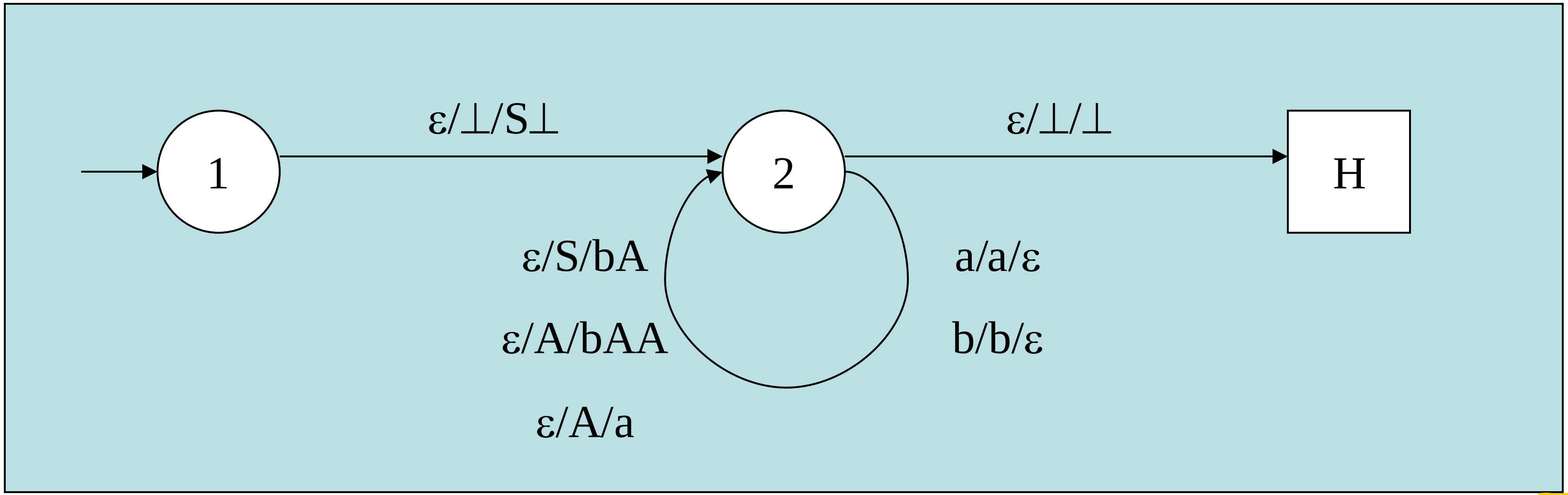
Constructing a PDR

- For each production **X → Z** we have an arc from state 2 to 2 labelled ε/X/Z
  - – e.g. S → bA   A → bAA | a



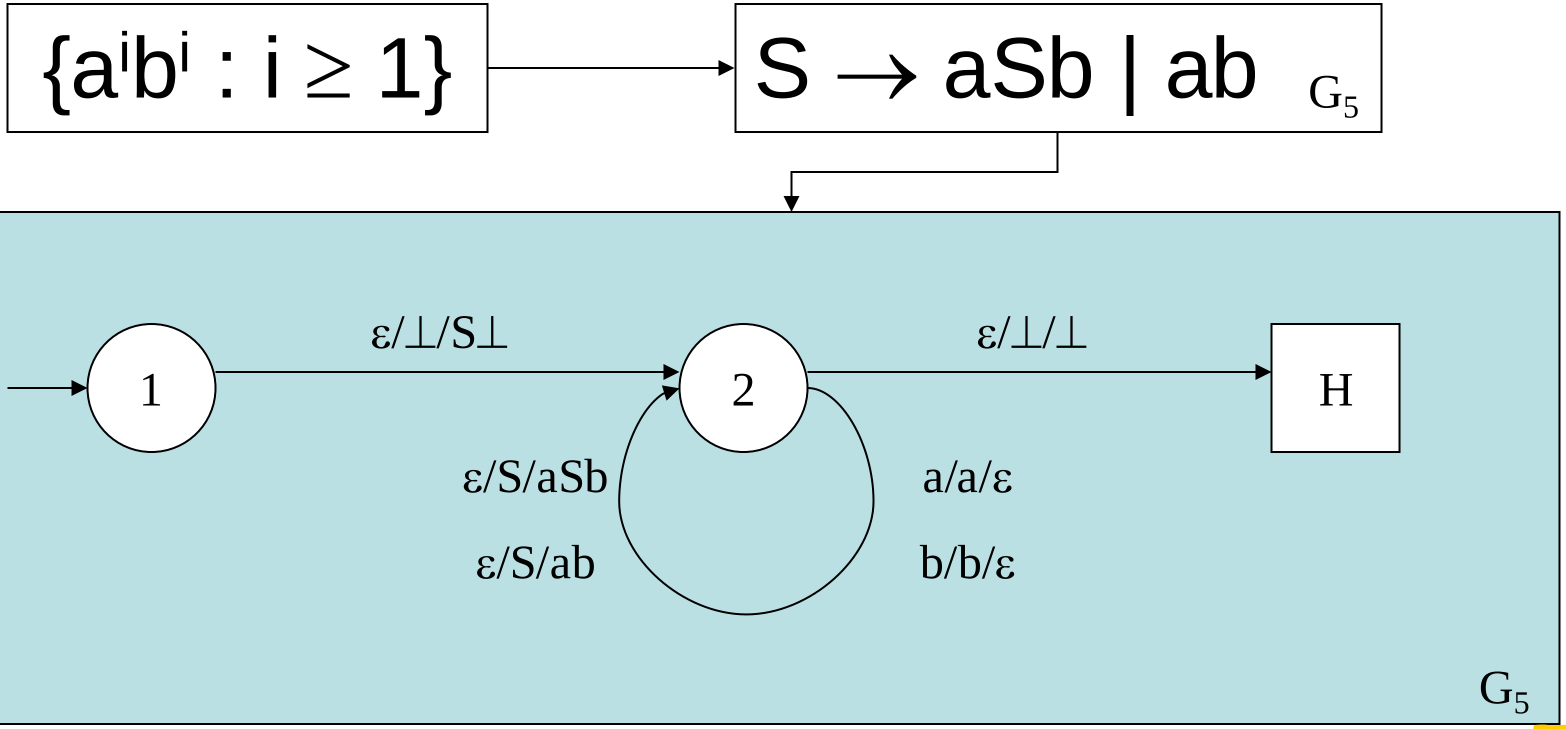
Constructing a PDR

- For each terminal (t) in the grammar we have an arc from state 2 to 2 labelled t/t/ε
  - e.g. S → bA, A → bAA | a



Constructing a PDR for G_5

- Lets look again at our non-regular language



$\{a^i b^i : i \geq 1\}$

$S \rightarrow aSb \mid ab$ G_5

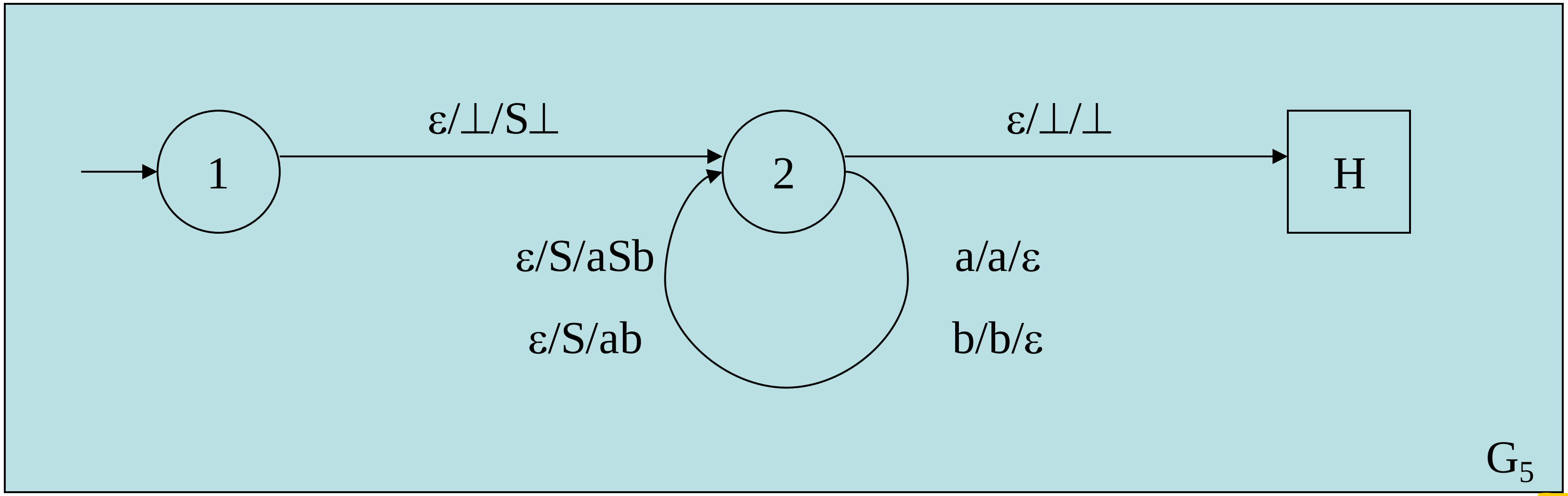
How to … (summary)

1. we have three states - a start state 1, a state 2, and a finish state H
2. we have an arc from 1 to 2 labelled “ε / $\perp$ / S $\perp$” (where S is the distinguished symbol)
3. we have an arc from 2 to H labelled “ε / $\perp$ / $\perp$”
4. for each production X → s we have an arc from 2 to 2 labelled “ε / X / s”
5. for each terminal t in the grammar we have an arc from 2 to 2 labelled “t / t / ε”

   – here the ε means “push nothing on the stack”

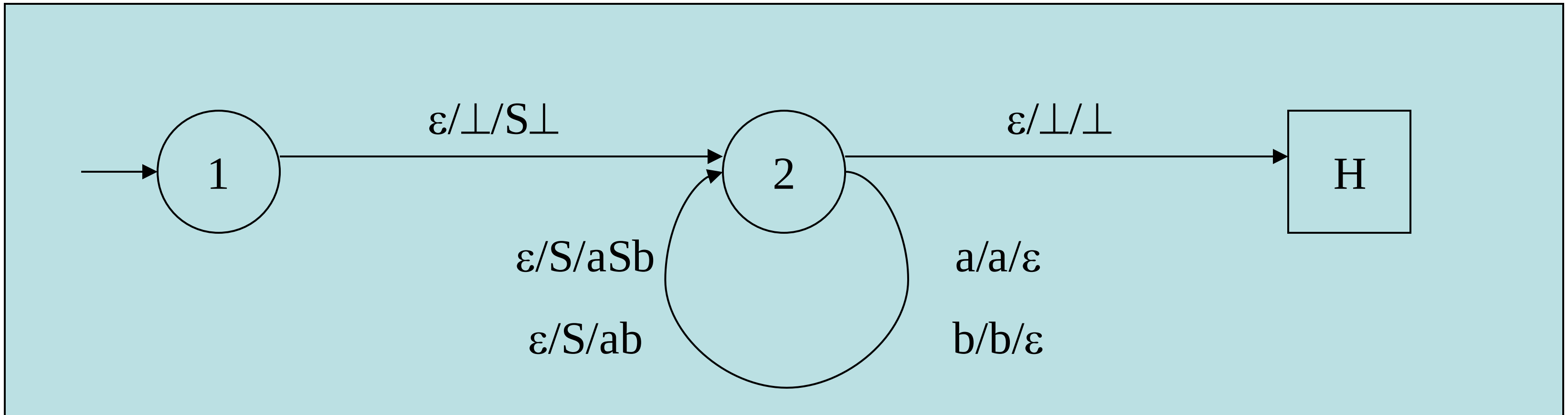
Parsing With the PDR for G_5

- We can now try to parse with our PDR
  - – We can try *aaabbb* and *aaabb*



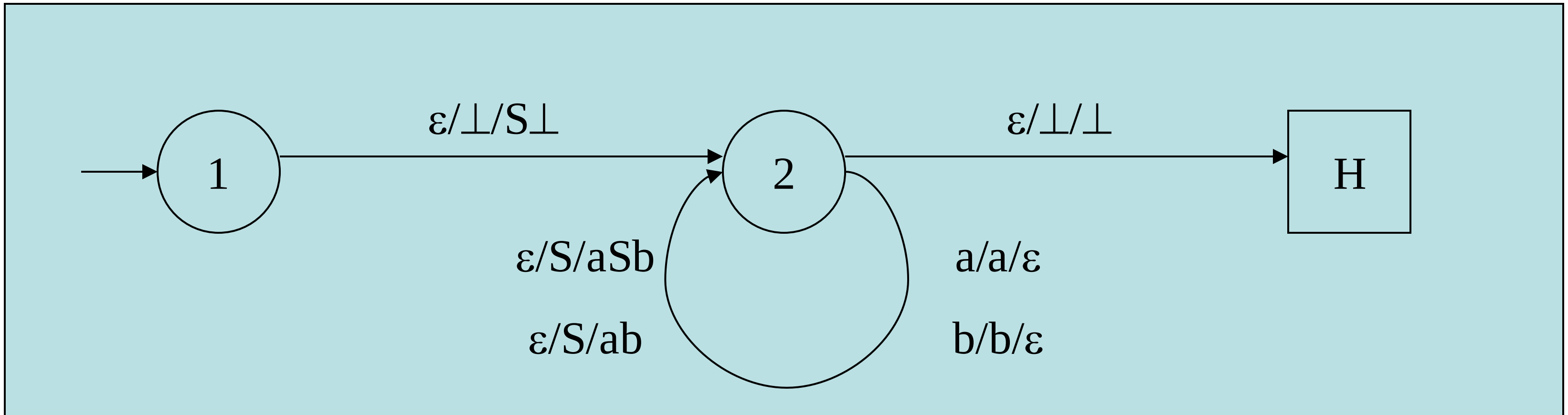
Using the PDA on *aaabbb*

- start in state 1 looking at a with ⊥ on the stack
- in state 2 looking at a with S⊥ on the stack
- in state 2 looking at a with aSb⊥ on the stack
- in state 2 looking at (the 2nd) a with Sb⊥ on the stack
- in state 2 looking at (the 2nd) a with aSbb⊥ on the stack
- in state 2 looking at (the 3rd) a with Sbb⊥ on the stack
- in state 2 looking at (the 3rd) a with abbb⊥ on the stack



Using the PDA on *aaabbb*

- in state 2 looking at b with bbb⊥ on the stack
- in state 2 looking at (the 2nd) b with bb⊥ on the stack
- in state 2 looking at (the 3rd) b with b⊥ on the stack
- in state 2 looking at end of input with ⊥ on the stack
- in state H looking at end of input with ⊥ on the stack



Parsing With a PDR

- A PDR is effectively doing a derivation on the stack from the start symbol S
- A string is valid if it can get to H with no input string left
- and the terminals it has generated in the derivation match from left to right against the input string

https://pixabay.com/photos/spaghetti-pasta-food-restaurant-863304/