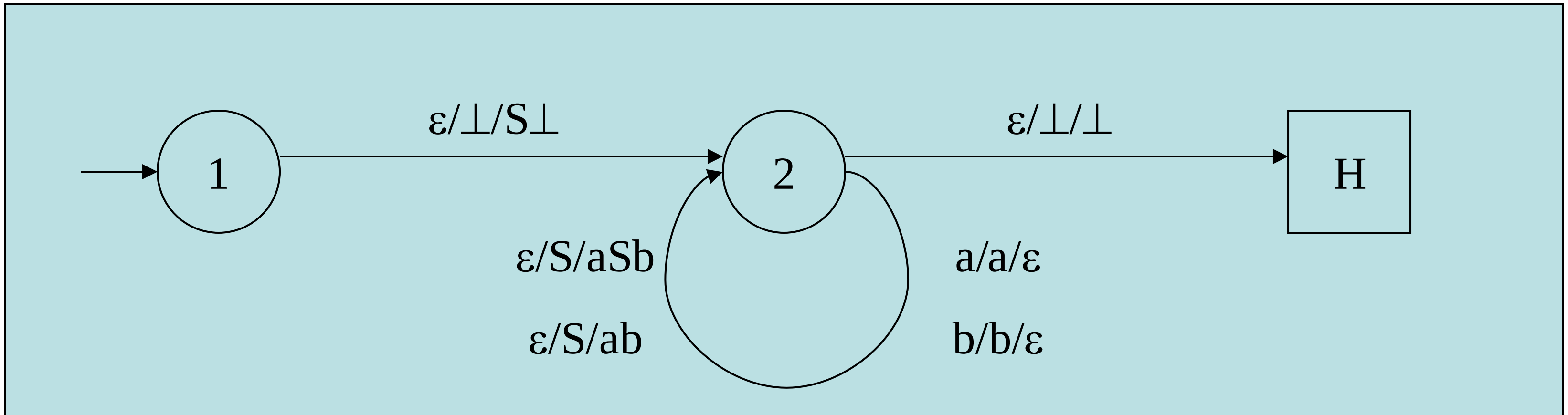
https://pixabay.com/photos/spaghetti-pasta-food-restaurant-863304/

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 terms it has generated in the derivation match from left to right against the input string

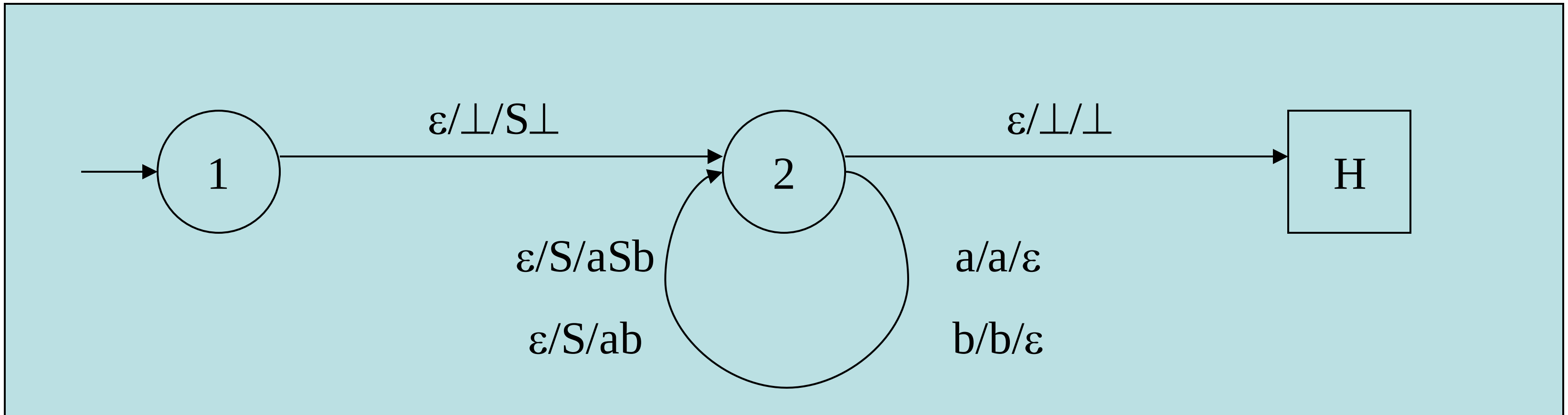
Using the PDA on *aaabb*

- 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

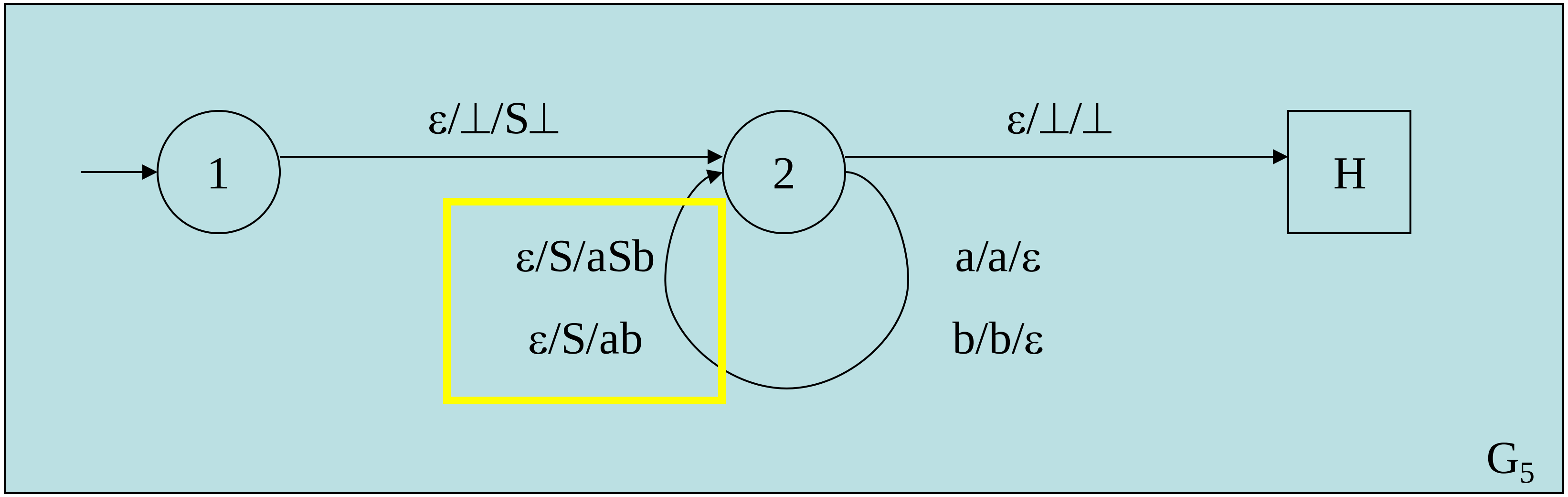


Using the PDA on *aaabb*

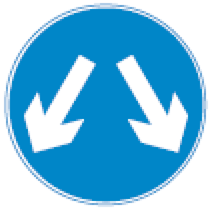
- in state 2 looking at b with bbb⊥ on the stack
- in state 2 looking at (the 2nd) b with bb⊥ on the stack
- in state 2 looking at end of input with b⊥ on the stack
- none of the arcs apply - we are stuck



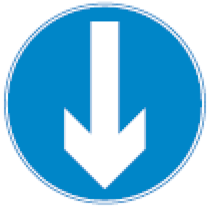
Non-determinism (again!)



- This type of PDR is non-deterministic



- Can we revise the PDR to make it deterministic?

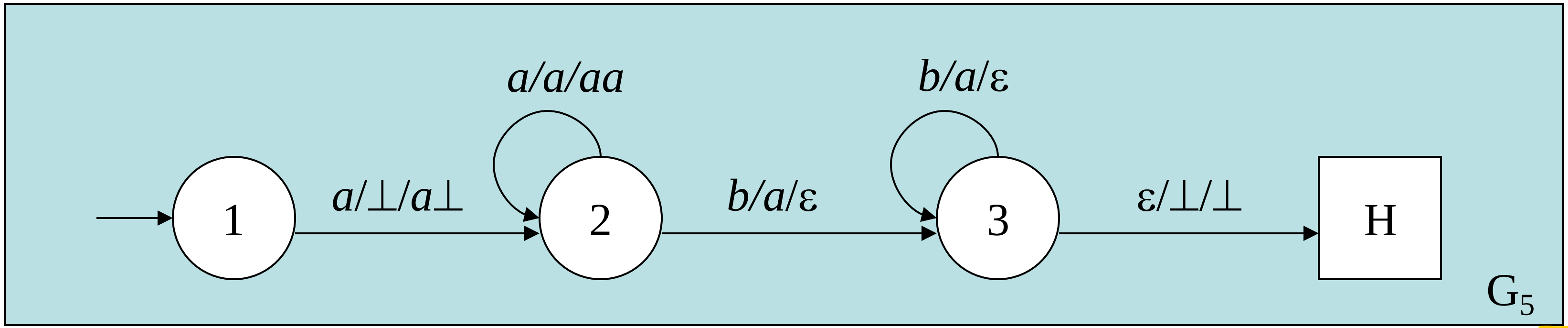


Converting to Deterministic PDR

- There is no formula for converting a non-deterministic PDR into a deterministic one
- The general rule of thumb is:
  - for every terminal (t) that can start the string there is an arc labelled t/⊥/t⊥
  - for numerical relations between characters (e.g. $a^i b^i$) push on *a* and pop off an *a* for every *b*
  - ends with ε/⊥/⊥ as usual

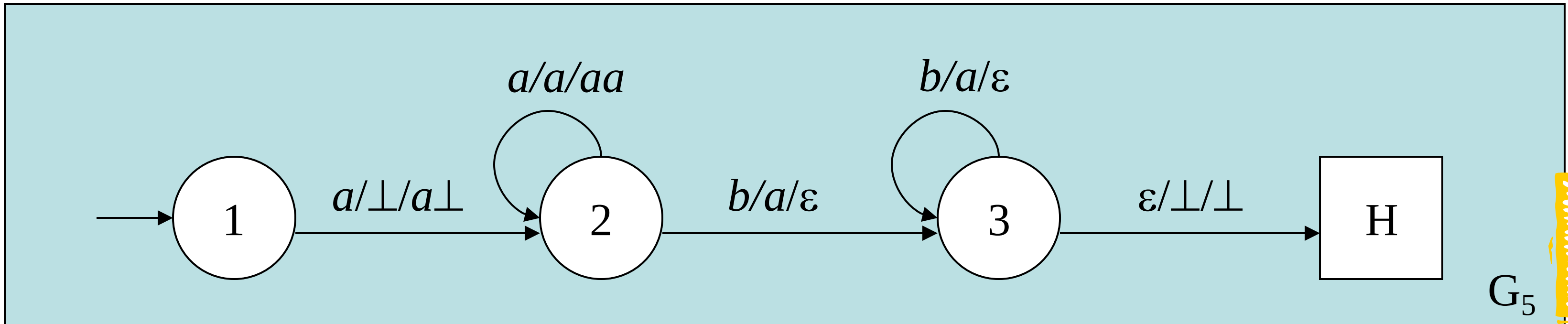
A Deterministic PDR for G_5

- G_5 is $\{a^i b^i : i \geq 1\}$
  - The input symbol separates the arcs from state 2 and the stack bottom marker separates the arcs from state 3
  - We can try *aaabbb* and *aaabb*



Using the New PDA on *aaabbb*

- start in state 1 looking at a with ⊥ on the stack
- in state 2 looking at the 2nd a with a⊥ on the stack
- in state 2 looking at the 3rd a with aa⊥ on the stack
- in state 2 looking at the b with aaa⊥ on the stack
- in state 3 looking at the 2nd b with aa⊥ on the stack
- in state 3 looking at the 3rd b with a⊥ on the stack
- in state 3 looking at end of string with ⊥ on the stack
- in state H looking at end of string with ⊥ on the stack



A Deterministic PDR for G_6

- Try these strings: *aabbaababb*, *babaa*, *abc*

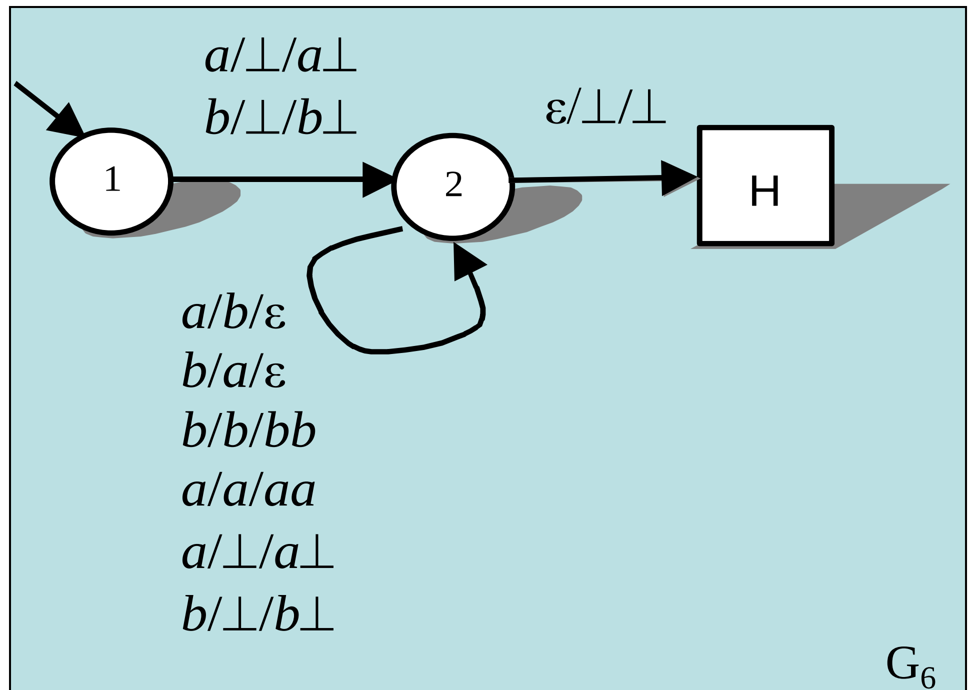
$S \rightarrow aB \mid bA \mid \varepsilon$

$A \rightarrow aS \mid bAA$

$B \rightarrow bS \mid aBB$

G_6

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



A Deterministic PDR for G_7

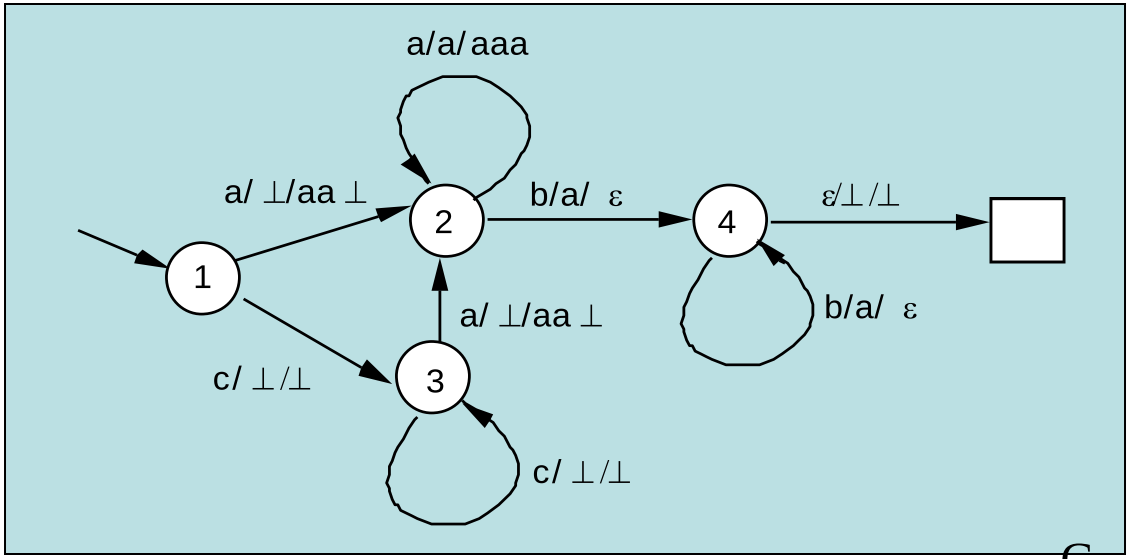
- Here the string may or may not start with *c*
  - Try: *ccccaabbb*, *abbb*, *caaabbbbbb*

$S \rightarrow cS \mid A$

$A \rightarrow aAbb \mid abb$

G_7

$\{c^i a^j b^{2j} : i \geq 0, j \geq 1\}$



G_7

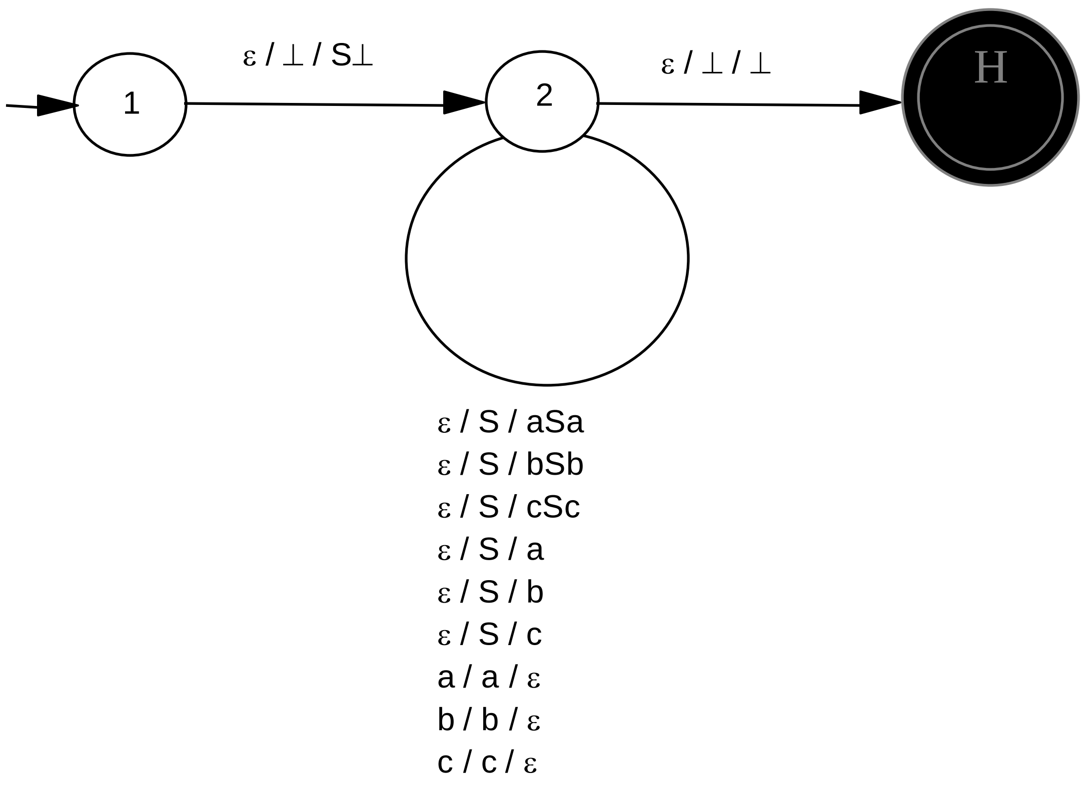
Special Non-Deterministic PDRs

- Consider the context free grammar (G_8):
  - $S \rightarrow aSa \mid bSb \mid cSc \mid a \mid b \mid c$
  - it generates palindrome strings
  - definitely a context free grammar, and hence the language is context free

- Can we design a deterministic PDR for the grammar G_8?
  - If not, why not?

A PDA for the Grammar G_8

– we could construct a (non-deterministic) PDA as in the earlier example



A Partial Stack Trace for This PDA Applied to abacbcaba

- 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 aSa⊥ on the stack
- in state 2 looking at b with Sa⊥ on the stack
- in state 2 looking at b with bSba⊥ on the stack
- in state 2 looking at a with Sba⊥ on the stack
- in state 2 looking at a with aSaba⊥ on the stack
- in state 2 looking at c with Saba⊥ on the stack
- in state 2 looking at c with cScaba⊥ on the stack
- in state 2 looking at b with Scaba⊥ on the stack

Now What?

- now shall we use the "ε / S / bSb" or the "ε / S / b" arc?
  - in fact there was a choice like this every time we rewrote S
- the only way to be sure is to see how many characters there are to the end of the string
- the problem is that the PDA has to "guess" where the middle of the palindrome is
- we cannot create a deterministic version of this PDA

Special Non-Deterministic PDRs

- We could try the following technique:
  - Push each symbol from the input stream onto the stack until the middle of the input string is reached, then pop a corresponding symbol off the stack for each of the remaining symbols in the input stream.
- But how does the PDR know when the middle of the input string has been reached?

Non-Deterministic CFLs

- Unlike regular languages, which are all deterministic, many context free languages are only non-deterministic
- Such languages cannot be processed by a deterministic PDR

Deterministic and Non-Deterministic CFLs

- so the set of all context-free languages is divided into
  - a set for which there is a deterministic PDA to parse them
  - a set for which there is no deterministic PDA (only a non-deterministic one)

Uses of Context Free Grammars

- with Regular Grammars everything is straightforward - we can create a deterministic FSM and parse with it

- with Context-Free Grammars things are not so straightforward - our CF grammar may be non-deterministic

But …

- Context Free Grammars are very important in computer science for compiling etc.
- The syntax of most programming languages are properly context free
  - e.g. the original version of Pascal
- How can we discover if a language is properly context free?
  - We can use LR(k) parsing
  - LR stands for "Look ahead Right"

Much more on this in the second half of SCC312. Web search for Yacc, Bison, etc

Chomsky Hierarchy: our roadmap

Type	Grammar	Machine	Other Equivalent
2	Context Free	Push Down Recogniser	
3	Regular	Finite State Recogniser	Regular Expressions

Non-deterministic only/deterministic

Ambiguity

Syntax and Semantics

Reminder

- Languages usually have meaning (**semantics**) as well as structure (**syntax)**
- In natural languages, sentences can be syntactically correct but not make sense:
  - *“the raggedy doctor parsed the zarbi-oriented flowery tardis”*
- Moreover, natural languages can be highly **ambiguous** (more than one meaning)
  - *“Fruit flies like a banana”*
  - *“Doctor who saved my life”*

Ambiguity in CFLs

- Formally, ambiguity means that within a language one or more sentences can be parsed into more than one structure

Ambiguity in CFLs

- Consider the following non-deterministic language (G_9)

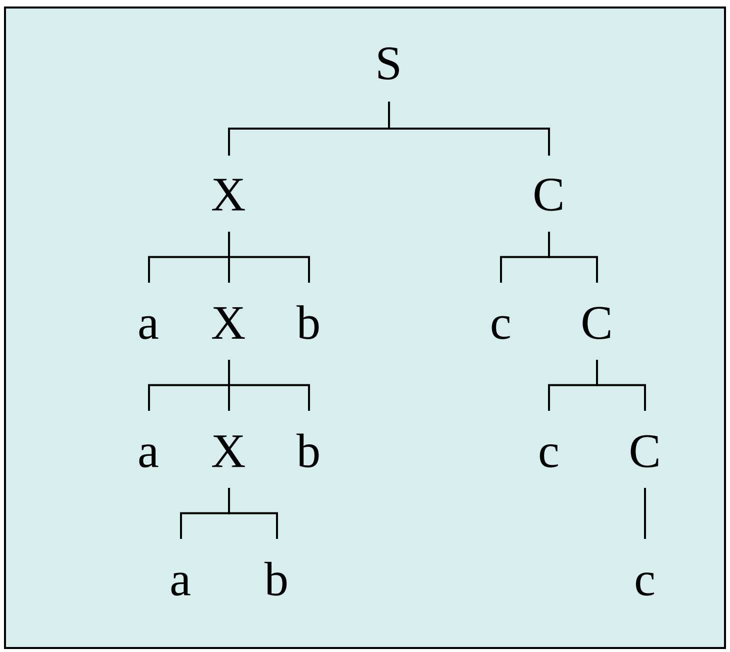
$\{a^i b^j c^k : i, j, k \geq 1, i = j \text{ or } j = k\}$ $\rightarrow$

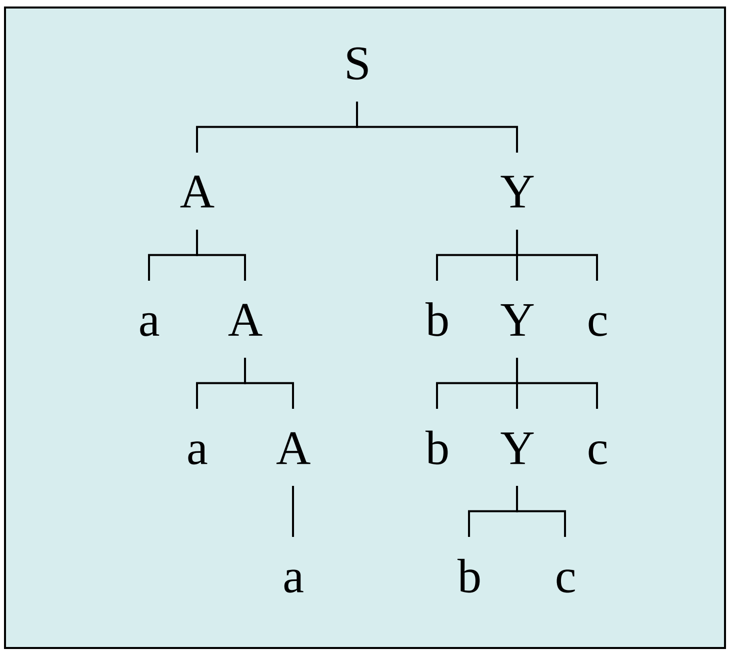
$$
\begin{aligned}
S &\rightarrow XC \mid AY \\
X &\rightarrow aXb \mid ab \\
Y &\rightarrow bYc \mid bc \\
C &\rightarrow cC \mid c \\
A &\rightarrow aA \mid a
\end{aligned}
$$

G_9

Ambiguity in G_9

- Any sentence of the form $a^i b^i c^i$ will be associated with two derivation trees:





A Proposed Grammar for (Simple) Arithmetic Expressions

- <expression> ::= <factor> |
  <expression> + <expression> |
  <expression> - <expression> |
  <expression> * <expression> |
  <expression> / <expression>
- <factor> ::= number |
  identifier |
  (<expression>)

e.g. an expression
something like:
“2 + 3 * b – a / 2”

Backus-Naur Form (BNF)

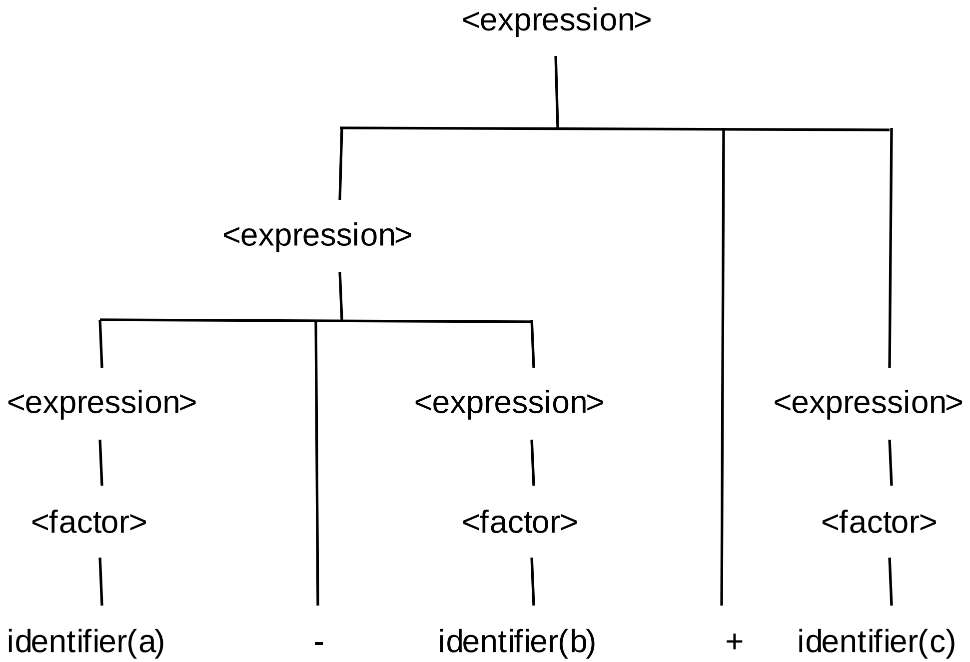
Reminder

- non-terminals in < ... > brackets
- alternatives for the same non-terminal are written as a single right-hand side, separated by | (meaning “or”)

Deriving a - b + c

- <expression>
- <expression> + <expression>
- <expression> + <factor>
- <expression> + identifier (c)
- <expression> - <expression> + identifier (c)
- <expression> - <factor> + identifier(c)
- <expression> - identifier(b) + identifier(c)
- <factor> - identifier(b) + identifier(c)
- identifier(a) - identifier(b) + identifier(c)

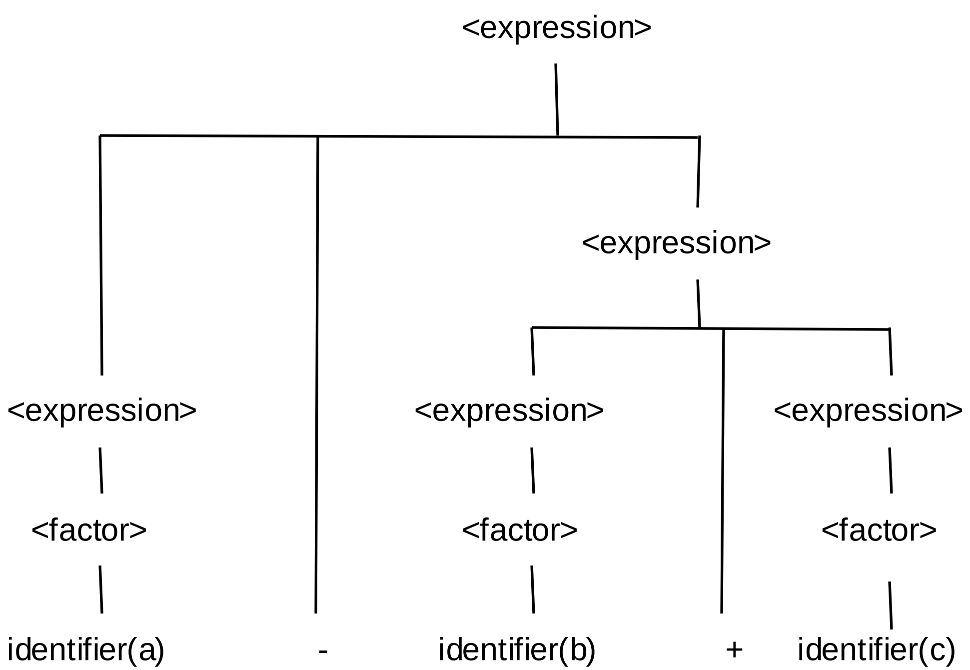
The Parse Tree for a - b + c



Another Derivation of a - b + c

- <expression>
- <expression> - <expression>
- <factor> - <expression>
- identifier(a) - <expression>
- identifier(a) - <expression> + <expression>
- identifier(a) - <factor> + <expression>
- identifier(a) - identifier(b) + <expression>
- identifier(a) - identifier(b) + <factor>
- identifier(a) - identifier(b) + identifier(c)

Another Parse Tree for a - b + c



Ambiguous Grammars I

- this grammar (let's call it G1) gives rise to two possible parses for this (and other) strings - there is an ambiguity
- the ambiguity is important - if a = 6, b = 4 and c = 5 the first parse of "a-b+c" evaluates to (6-4)+5 or 7, and the second to 6-(4+5) or -3

Ambiguous Grammars II

- a sentence (grammatical string) is **ambiguous** if it can be parsed according to a grammar in at least two different ways (that is, the parse trees are different, not just the order of derivation)
- a grammar is **ambiguous** if there is at least one ambiguous sentence according to the grammar

An Alternative Grammar (G2) I

- <expression> ::= <factor> |
  <expression> + <factor> |
  <expression> - <factor> |
  <expression> * <factor> |
  <expression> / <factor>
- <factor> ::= number |
  identifier |
  (<expression>)

The right hand sides are new

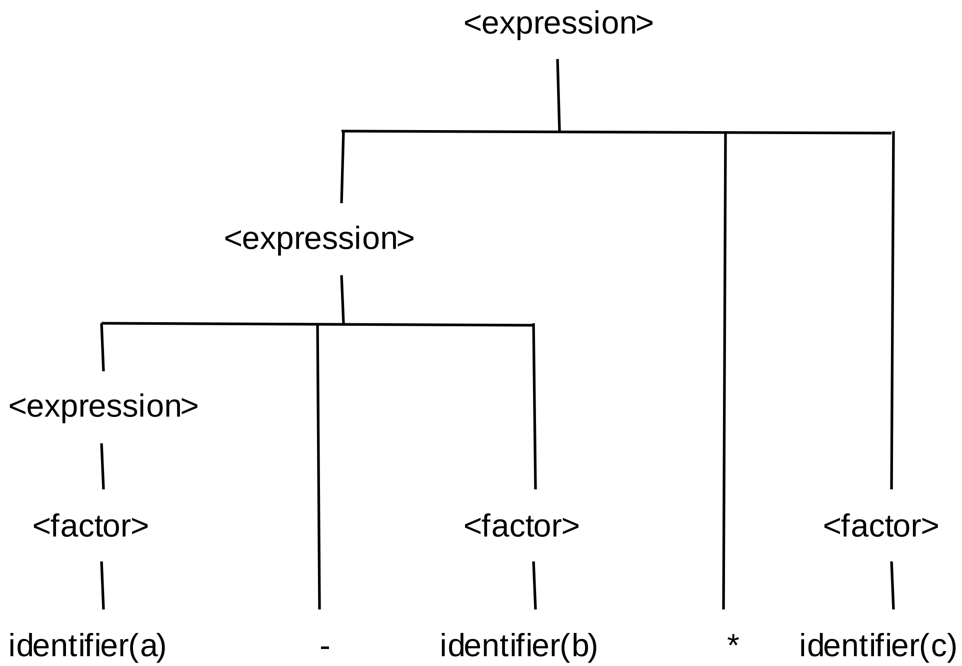
An Alternative Grammar (G2) II

- this grammar agrees with G1 as to which strings are grammatical and which are not - that is, the grammars are (weakly) **equivalent**
- but grammar G2 disallows the second parse tree - check that you can see why
- this appears to be what we want, but there is still a problem

Parsing a - b * c with Grammar G2

- <expression>
- <expression> * <factor>
- <expression> * identifier(c)
- <expression> - <factor> * identifier(c)
- <expression> - identifier(b) * identifier(c)
- <factor> - identifier(b) * identifier(c)
- identifier(a) - identifier(b) * identifier(c)

The Parse Tree for a - b * c



Precedence

- so grammar G2 parses a - b * c as if it was (a - b) * c
- however, the usual convention is that the operators * and / have higher **precedence** than + and -
- so a - b * c should be interpreted as a - (b * c), even if the programmer doesn’t insert the brackets
- so let’s try again

A Third Attempt - Grammar G3

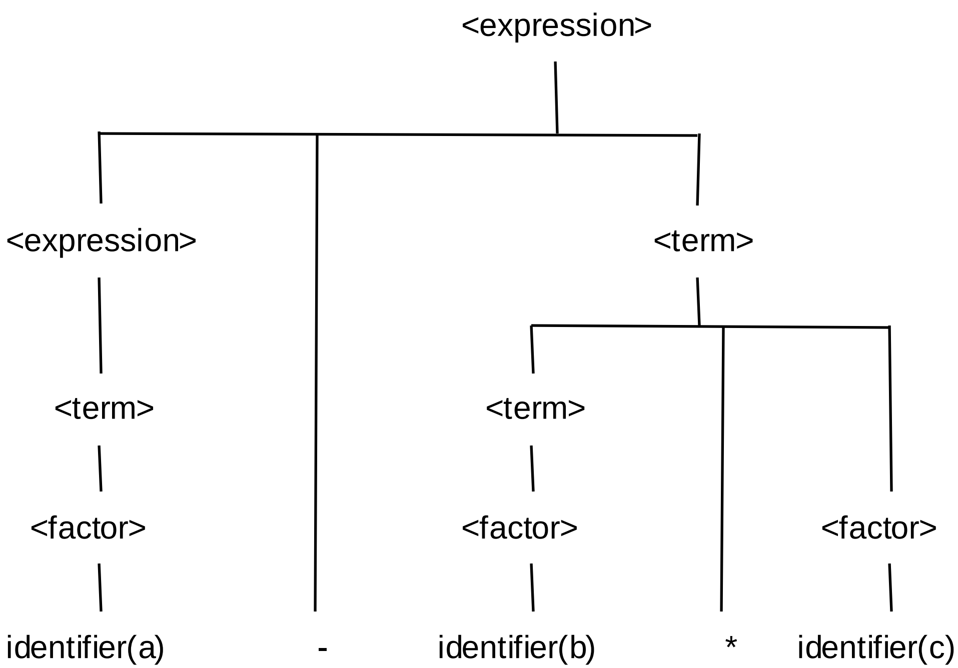
- <expression> ::= <term> |
  <expression> + <term> |
  <expression> - <term>
- <term> ::= <factor> |
  <term> * <factor> |
  <term> / <factor>
- <factor> ::= number |
  identifier |
  (<expression>)

This is the bit that differs from G2

Parsing a - b * c with Grammar G3

- <expression>
- <expression> - <term>
- <expression> - <term> * <factor>
- <expression> - <term> * identifier(c)
- <expression> - <factor> * identifier(c)
- <expression> - identifier(b) * identifier(c)
- <term> - identifier(b) * identifier(c)
- <factor> - identifier(b) * identifier(c)
- identifier(a) - identifier(b) * identifier(c)

The Parse Tree for a - b * c



Grammar G3

- can be shown to be **unambiguous** (that is, there is only one way of parsing any particular valid string)
- is (weakly) equivalent to grammar G1
- gives the correct **precedence** to the arithmetic operators
  - for instance, there is no way to parse “a - b * c” as if it had the structure “(a - b) * c” (unless you explicitly write the brackets)

Ambiguity in Programming

- When compiling source code programs, the compiler generates a parse tree of the statements
  - the syntactic structure is used as a basis for the generation of the code
  - if there are two possible syntactic structures for a statement, there are two possible ways in which a statement could execute.
  - a program could execute in ways we do not expect

Ambiguity in Pascal

- Now consider the following fragment from the original definition of Pascal:

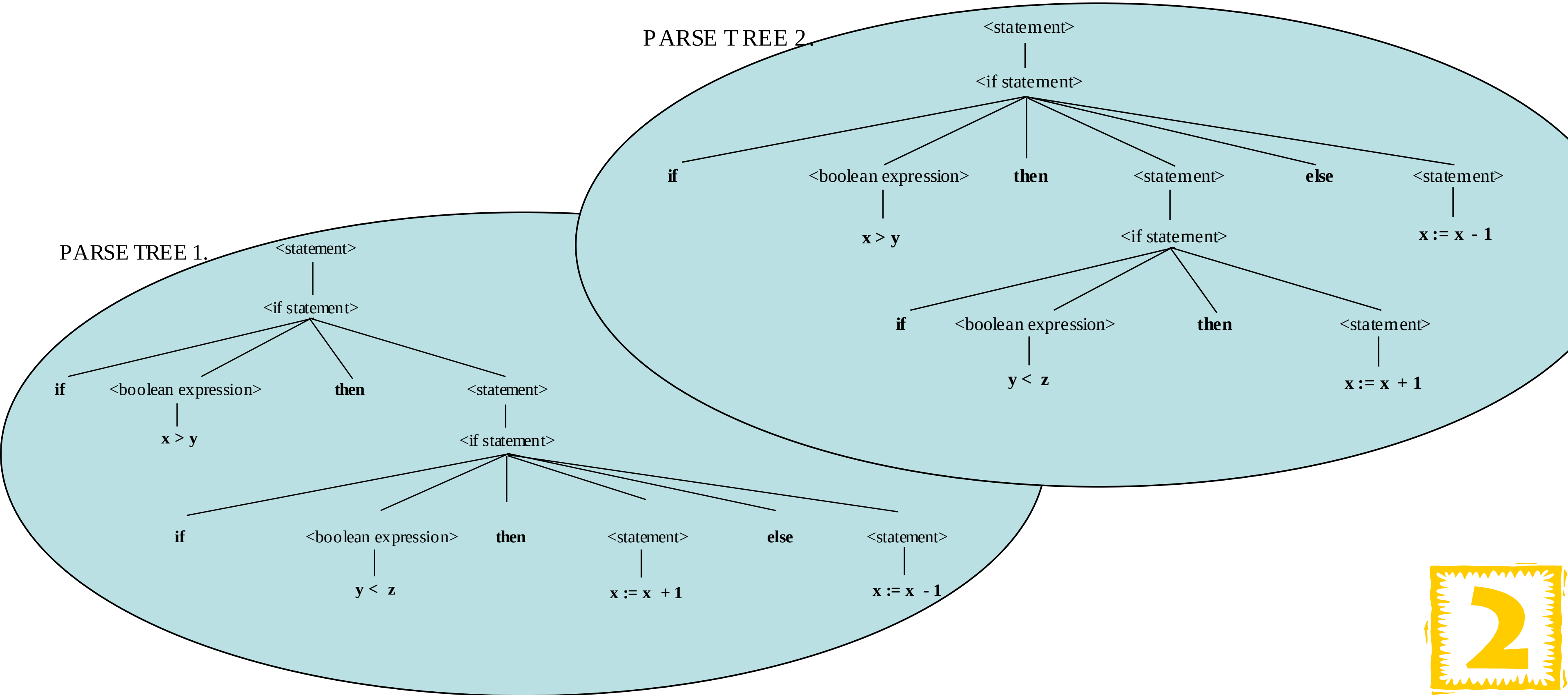
  <statement> ::= <if statement> |
  <assignment statement> | …

  <if statement> ::=
  if <boolean expression> **then** <statement> |
  if <boolean expression> **then** <statement> **else** <statement>

- The above definition is ambiguous

Ambiguity in Pascal

- We can generate two parse trees for:
  - if x > y then if y < z then x := x + 1 else x := x - 1



Ambiguity in Pascal

- Consider the effect on the execution of:
  - if x > y then if y < z then x := x + 1 else x := x – 1
  - where x = 2, y = 1, and z = 0
  - Parse Tree 1:
    - if 2 > 1 then (if 1 < 0 then x := 2 + 1 else x := 2 – 1)
    - x := 1
  - Parse Tree 2:
    - if 2 > 1 then (if 1 < 0 then x := 2 + 1) else x := 2 – 1
    - Nothing changes

Semantic Implications

- In itself, formal ambiguity is not necessarily a problem
  - what matters is if the semantics of two syntactic structures for the same statement are different
  - these are the **semantic implications**
  - this is the case in the Pascal example and the order of precedence example

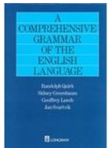
Ambiguity in Grammars

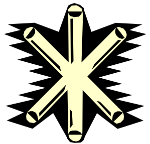
- Ambiguity in grammars for programming languages is not acceptable
- We would like to rewrite any ambiguous grammars to become unambiguous
- It turns out that this is not always possible
- Some context free languages are **inherently ambiguous**

Chomsky Hierarchy

Type	Grammar	Machine	Other Equivalent
2	Context Free	Push Down Recogniser	
3	Regular	Finite State Recogniser	Regular Expressions

Non-deterministic only/deterministic





Summary

Week 13

- Not all languages are regular, as can be shown by the "Repeat State" Theorem

For further reading, web search "pumping lemma"

- Context-free grammars have rules: **X → RHS**
- For every CF grammar there is a pushdown recogniser, but not all are deterministic

- CF grammars are used for most programming languages
- Ambiguous grammars have semantic implications

https://pixabay.com/photos/youth-active-jump-happy-sunrise-570881/