CS 320: Principles of Programming Languages

Mark P Jones, Portland State University

Spring 2019

Week 2: Describing Syntax

language = { syntax, semantics }

syntax: the written/spoken/symbolic/physical form; how things are communicated

language = { syntax = { concrete, abstract }, semantics }

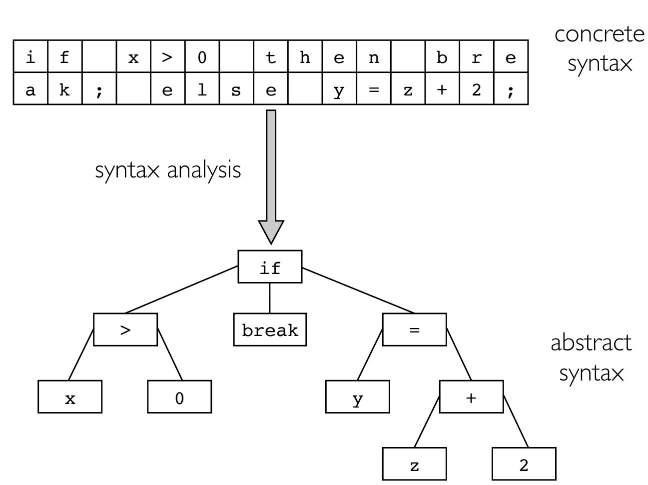
concrete syntax: the representation of a program text in its source form as a sequence of bits/bytes/characters/lines

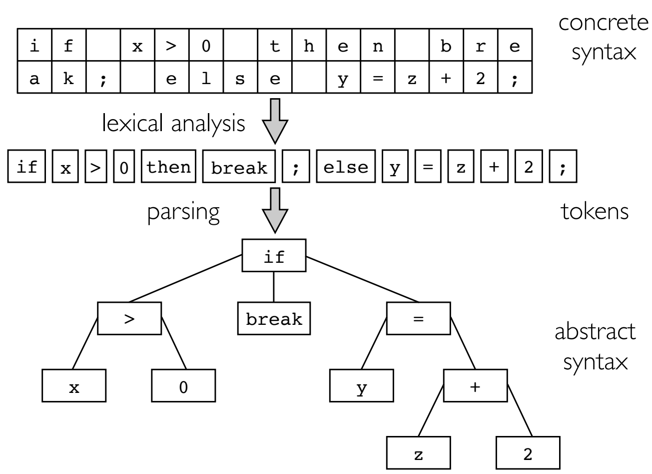
language = { syntax = { concrete, abstract }, semantics }

abstract syntax: the representation of a program structure, independent of written form

language = { syntax = { concrete ⇩ abstract }, semantics }

syntax analysis: This is one of the areas where theoretical computer science has had major impact on the practice of software development





Wanted!

- We want methods for describing language syntax that are:
  - clear, precise, and unambiguous
  - expressive (e.g., finite descriptions of infinite languages)
  - suitable for use in the implementation of practical syntax analysis tools (lexical analyzers, parsers, …)
- Formal language theory provides a rigorous foundation for specifying syntax. In particular:
  - Regular languages are well-suited to describing lexical syntax (what sequences of characters are valid tokens?)
  - Context-free languages are well-suited to describing grammatical structure (what sequences of tokens are valid sentences?)

Formal languages

- Pick a set, A, of **symbols**, which we refer to as the **alphabet**

  For lexical analysis, "symbols" are typically characters

  For parsing, "symbols" are typically tokens
- The set of all finite strings of symbols taken from A is written as A*
- A **language** (over A) is a subset of A*

Examples

- If A = { 0, 1 }, then:

  A* = { "", "0", "1", "00", "01", "10", "11", "000", ... }
- Bytes form a finite language over A:

  Bytes = { $b_0\, b_1\, b_2\, b_3\, b_4\, b_5\, b_6\, b_7 \mid b_i \in \{0,1\}$ }
- Even length bitstreams form an infinite language over A

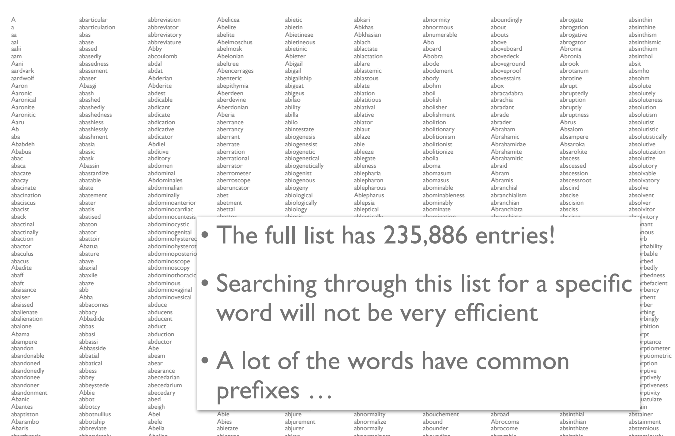
  Evens = {"", "00", "01", "10", "11"} ∪ { x y | x, y ∈ Evens }

  string concatenation

English as a formal language

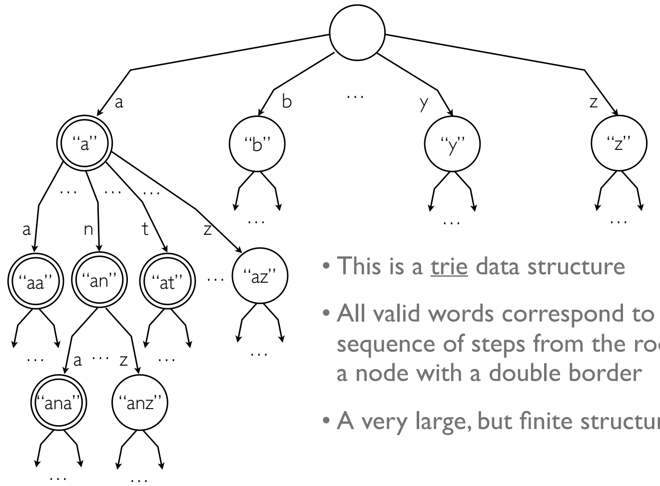
- With alphabet A = { "a", "b", "c", ... , "z" }:
  - A* = all strings with zero or more letters from A
  - Words = the subset of A* containing only those strings that are valid words in English e.g., "language" ∈ Words, "jubmod" ∉ Words
  - English = the subset of Words* containing only those strings that are valid sentences in English ["languages", "are", "interesting"] ∈ English
- But how do we specify which subsets to use?

A list of English words (`/usr/share/dict/words`)



- The full list has 235,886 entries!
- Searching through this list for a specific word will not be very efficient
- A lot of the words have common prefixes …

Let's give this another trie …



- This is a trie data structure
- All valid words correspond to a sequence of steps from the root to a node with a double border
- A very large, but finite structure

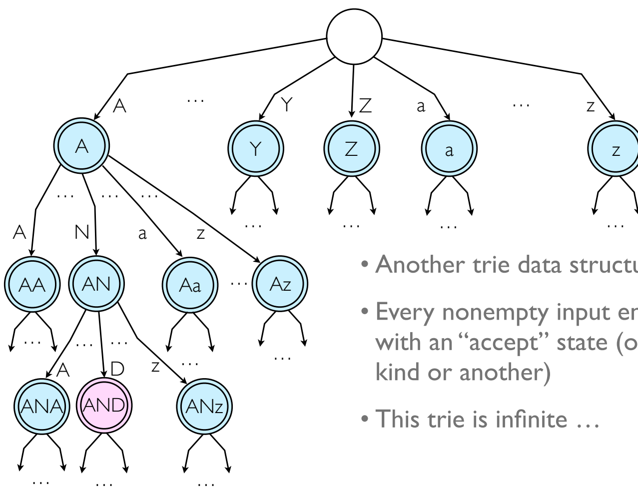
Prop as a formal language

- With alphabet A = { "A", "B", "C", … , "Z", "(", ")", … }:

A*	=	all strings with zero or more letters from A	
Tokens	=	Keywords:	"AND", "OR", "NOT";
		Literals:	"TRUE", "FALSE";
		Punctuation:	"(" and ")"
		Variable names:	all nonempty subsets of A* containing only letters
Prop	=	the subset of Tokens* corresponding to valid circuits	

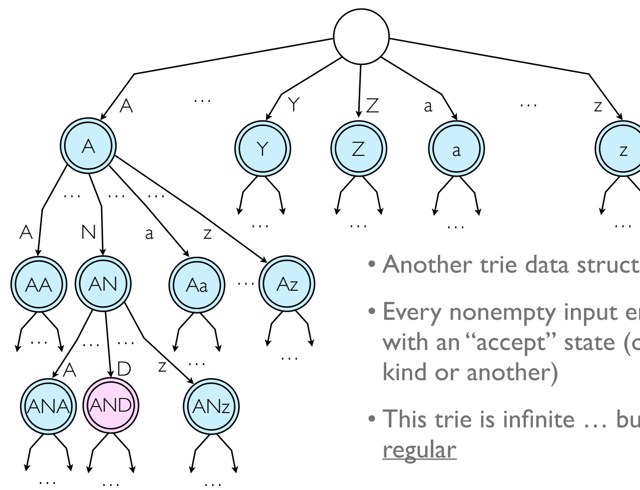
- But how do we specify these details?

Tokens in Prop (a subset)



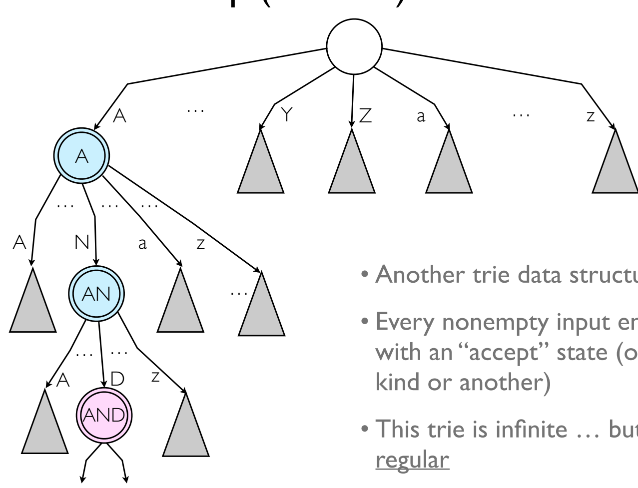
- Another trie data structure
- Every nonempty input ends with an "accept" state (of one kind or another)
- This trie is infinite …

Tokens in Prop (a subset)



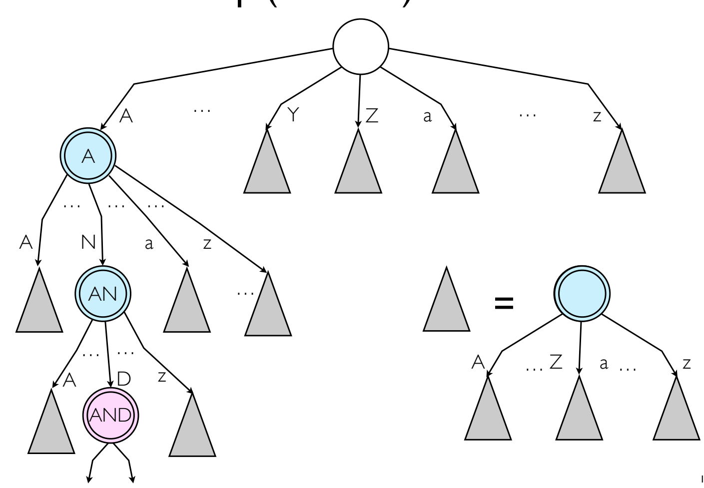
- Another trie data structure
- Every nonempty input ends with an "accept" state (of one kind or another)
- This trie is infinite … but regular

Tokens in Prop (a subset)

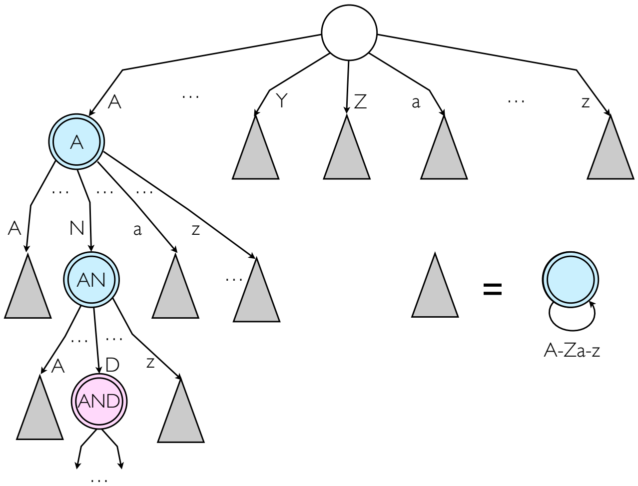


- Another trie data structure
- Every nonempty input ends with an "accept" state (of one kind or another)
- This trie is infinite … but regular

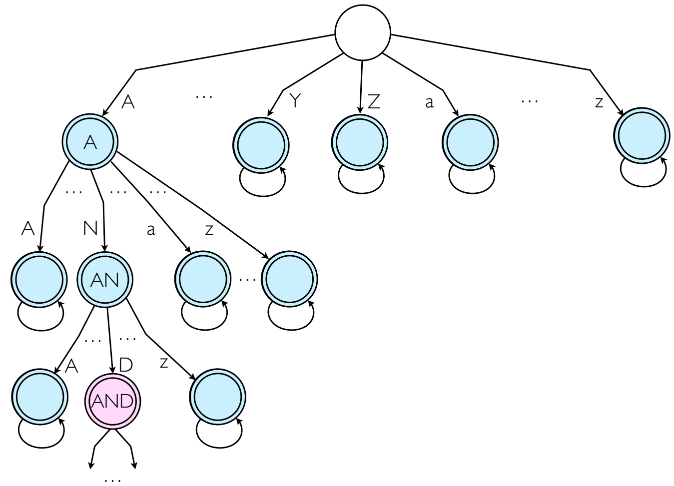
Tokens in Prop (a subset)



Tokens in Prop (a subset)

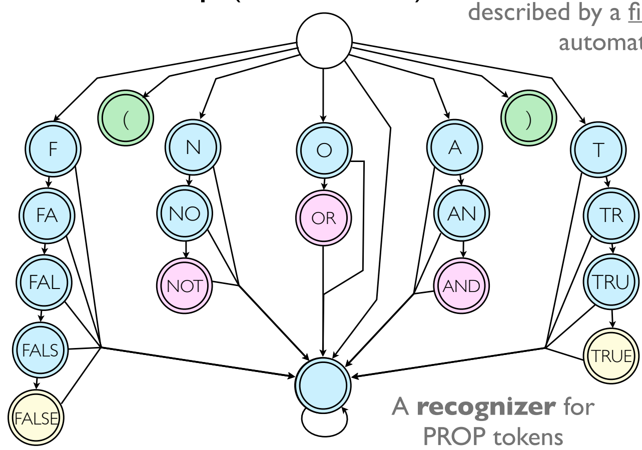


Tokens in Prop (a subset)



Tokens in Prop (all of them!)

An <u>infinite</u> language, described by a <u>finite</u> automaton!

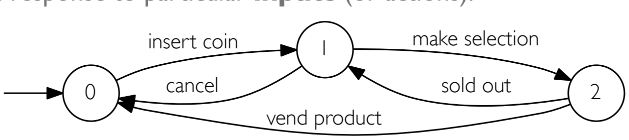


A **recognizer** for PROP tokens

Finite Automata and Regular Languages

Terminology

A **finite automaton** (or state machine) describes a system that can **transition** (or move) between a finite set of **states** in response to particular **inputs** (or actions).

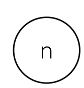


A **deterministic** finite automaton (DFA) has *at most one* way to transition out of any state in response to a given input

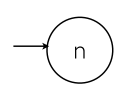
A **nondeterministic** finite automaton (NFA) may allow *multiple* transitions from some state in response to a given input

Finite automata building blocks

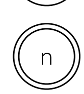
We use the following symbols to describe automata:



state, labeled n



start state

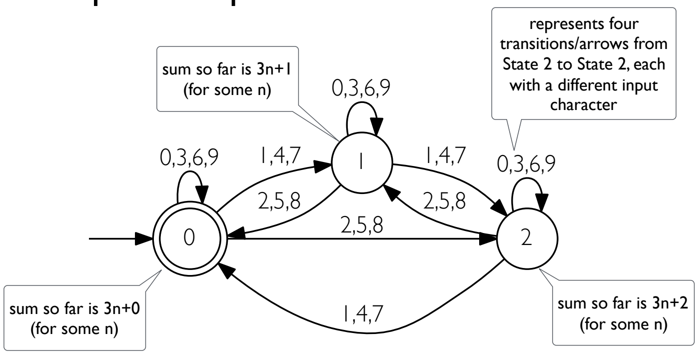


accept state

a transition on input a

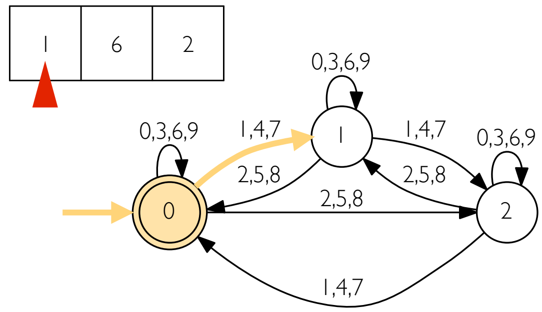
a transition without consuming any input, sometimes written: 

Example: multiples of 3



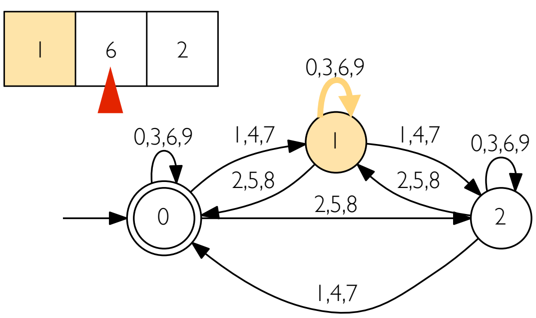
a number is divisible by three if the sum of its digits is divisible by three

Example: multiples of 3



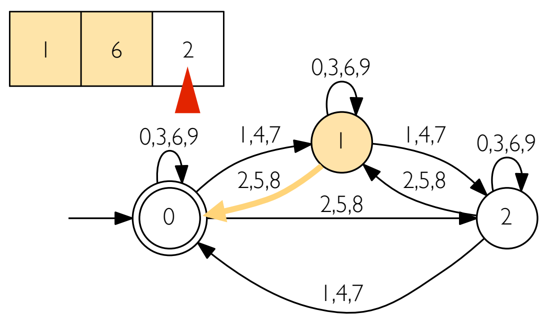
a number is divisible by three if the sum of its digits is divisible by three

Example: multiples of 3



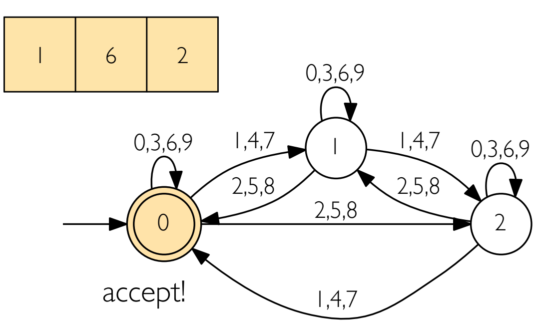
a number is divisible by three if the sum of its digits is divisible by three

Example: multiples of 3



a number is divisible by three if the sum of its digits is divisible by three

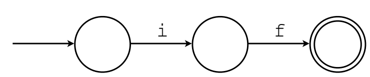
Example: multiples of 3



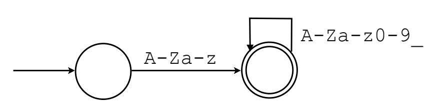
a number is divisible by three if the sum of its digits is divisible by three

Example: recognizers for lexical analysis

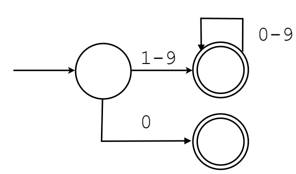
The "`if`" keyword:



Identifiers:

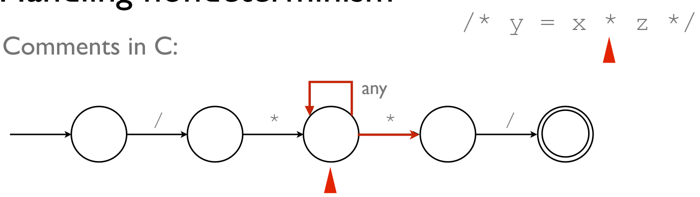


Integer literals: (without leading zeros)

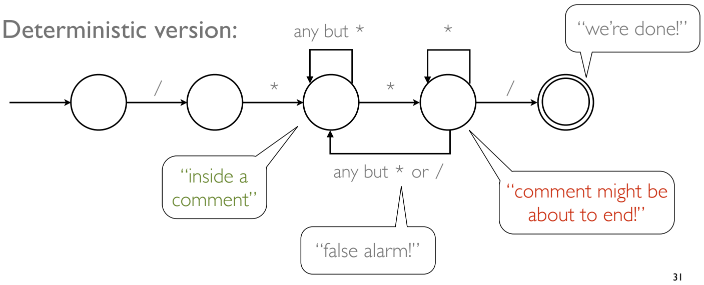


Handling nondeterminism

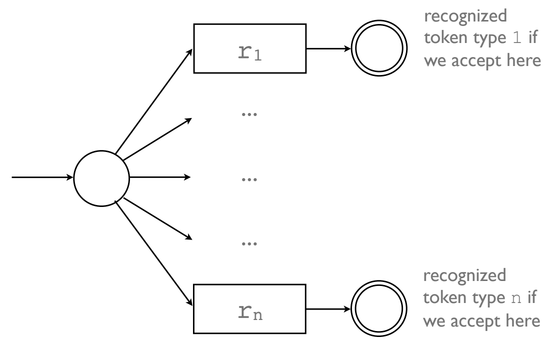
Comments in C:



Deterministic version:

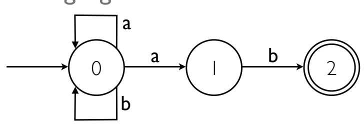


Combining multiple recognizers



Finite automata as language recognizers

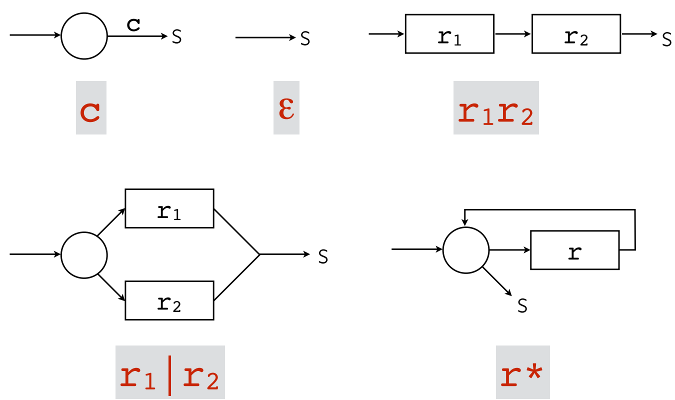
- Every finite automaton defines a formal language:
  - The set of strings corresponding to paths from the start state to an accept state
- Example: what language does this automaton describe?



Answer: {"ab", "aab", "bab", "aaab", "abab", "baab", "bbab", ...}

- However:
  - "a" is not included (does not reach an accept state)
  - "cab" is not included (gets stuck at state 0)

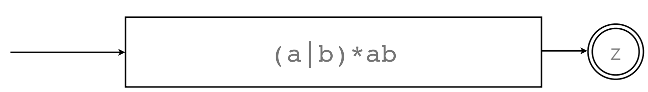
Capturing some recurring patterns



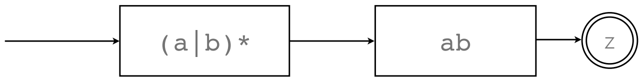
Regular expressions

- A widely used notation for describing patterns in text strings: emacs, vi, grep, awk, perl, ruby, python, javascript, …
- Also good for describing the lexical structure of programming languages ...
- Every regular expression corresponds to a DFA
- Every regular expression defines a language (the language that is recognized by the corresponding DFA)

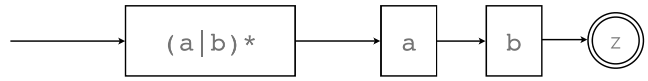
A recognizer for `(a|b)*ab`



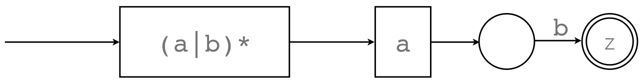
A recognizer for (a|b)*ab



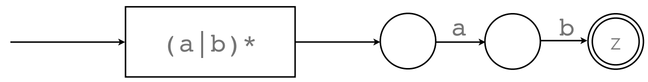
A recognizer for (a|b)*ab



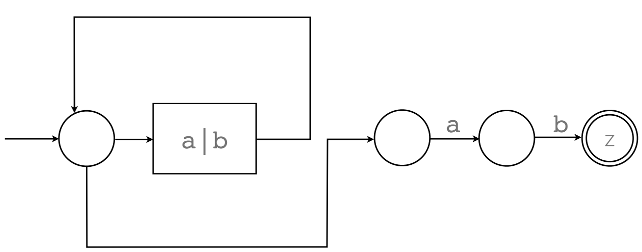
A recognizer for (a|b)*ab



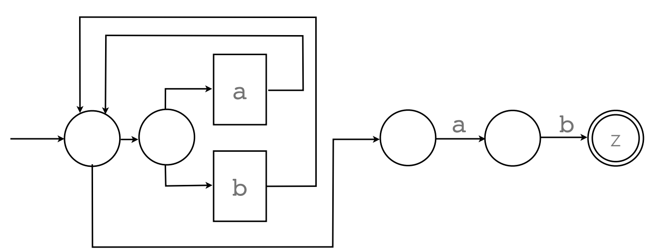
A recognizer for (a|b)*ab



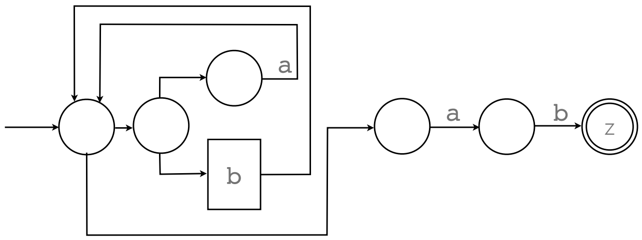
A recognizer for (a|b)*ab



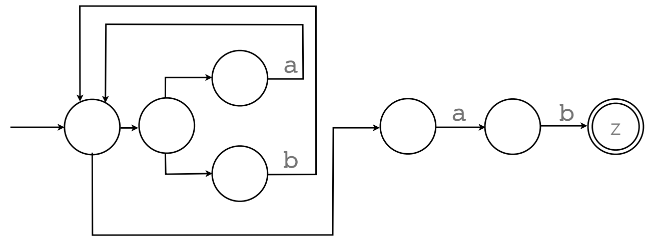
A recognizer for (a|b)*ab



A recognizer for `(a|b)*ab`

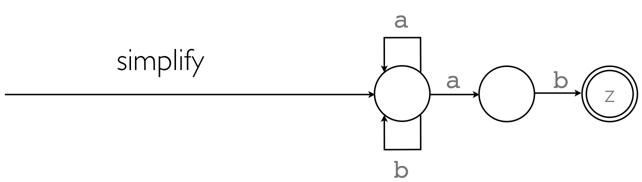


A recognizer for `(a|b)*ab`

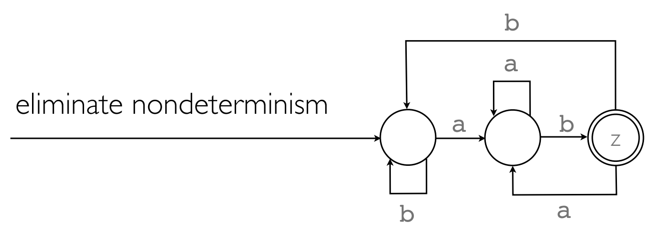


✓

A recognizer for `(a|b)*ab`



A recognizer for `(a|b)*ab`



✓

Regular expressions

Expression	Meaning
ε	**Empty**: matches the empty string
`c`	**Constant**: matches the single character `c`
$r_1 \mid r_2$	**Alternatives**: matches text matching r_1 or text matching r_2
$r_1\ r_2$	**Sequencing**: matches text matching r_1 followed by text matching r_2
`r*`	**Repetition**: matches a sequence of zero or more items, each of which matches `r`
`(r)`	**Grouping**: matches text matching `r`

Derived forms

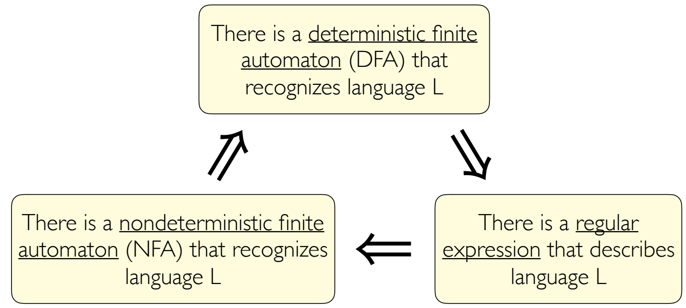
Expression	Meaning
`r+`	**Repetition**: a sequence of one or more items, each of which matches `r`. `r+` = `rr*`.
`r?`	**Optional**: optional text matching `r`. `r?` = `(r` | ε`)`.
`[abc]`	**Character classes**: short for `(a|b|c)` Also allows ranges of characters. e.g., `[a-zA-Z]`.
`.`	**Wildcard**: matches any character (except newline)

Programming language examples

Regular expression	Describes
`if`	the keyword **if**
`[A-Za-z][A-Za-z0-9_]*`	identifiers
`0|[1-9][0-9]*`	integer literals (without leading zeros)
`[ \t\n]*`	whitespace

`{int}{frac}?{exp}?`
`where {int}  = 0|[1-9][0-9]*`
`{frac} = \.{int}`
`{exp}  = [Ee](\+|-){int}`
	floating point literals
	(using auxiliary definitions)

Key things to know (see CS 311 for proofs, etc.)



In any/all of these situations, we say that L is a **regular language**

Wanted! (reprise)

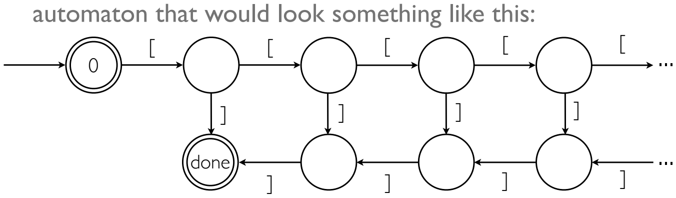
- We want methods for describing language syntax that are:
  - clear, precise, and unambiguous
  - expressive (e.g., finite descriptions of infinite languages)
  - suitable for use in the implementation of practical syntax analysis tools (lexical analyzers, parsers, …)
- Regular expressions work really well for describing lexical syntax …
- But the set of regular languages that they describe is quite limited …

A simple language of Brackets

- Brackets = { "" } ∪ { [b] | b ∈ Brackets }
- So the "words" in Brackets are:
  "", "[ ]", "[[ ]]", "[[[ ]]]", "[[[[ ]]]]", "[[[[[ ]]]]]", …
- In other words, nested pairs of bracket characters:
  - a sequence of n open brackets …
  - … followed by exactly n close brackets
- A subset of any language that uses parentheses/brackets
- But is it a regular language?

Brackets is not regular

- If brackets is regular, then we can recognize it using a **finite** automaton that would look something like this:



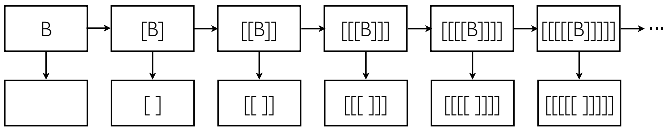
- If s_n is the state that we can reach after n open brackets and $n \neq m$, then $s_n \neq s_m$
- So this machine must have infinitely many states (at least one distinct state s_n for each n)
- So Brackets cannot be regular

Iteration vs recursion

- Regular expressions don't allow recursion, just iteration
- But it is easy enough to give a simple recursive characterization for B ∈ Brackets:

B ⟶	meaning: B is empty
B ⟶ [B]	meaning: B consists of an initial [symbol, another element from Brackets; and a closing] symbol

Generating brackets



- We have two rewrite rules:

  B $\longrightarrow$ replace a B with the empty string

  B $\longrightarrow$ [B]   replace a B with the string [B]

- Either rule can be used to rewrite an occurrence of B
- We say that "B **derives** s" if the string s can be obtained from B by repeated rewriting/replacement

Context-Free Grammars and Languages

Context-free grammars (CFGs)

A context-free grammar G = (T, N, P, S) consists of

- a set T of **terminal** symbols ("tokens")
- a set N of **nonterminal** symbols
- a set P of **productions**, each of which is a rule of the form n $\longrightarrow$ w where n $\in$ N, and w $\in$ (T $\cup$ N)*
- a **start symbol** S $\in$ N

Example

- A CFG for Brackets: ({ [,] }, { B }, { B $\longrightarrow$ ε, B $\longrightarrow$ [B] }, B)
- In practice, it is often sufficient just to write down the productions for a CFG:

  the sets of terminals and nonterminals can usually be inferred from the productions

  the start symbol can either be identified explicitly, or assumed as the first nonterminal

- For example, we can describe Brackets by: B $\longrightarrow$

  B $\longrightarrow$ [B]

More examples

Prop:
```
P ⟶ TRUE
P ⟶ FALSE
P ⟶ VAR
P ⟶ ( P )
P ⟶ AND P P
P ⟶ OR P P
P ⟶ NOT P
```

Arithmetic:
```
E ⟶ n
E ⟶ ( E )
E ⟶ E + E
E ⟶ E * E
```

Regexps:
```
R ⟶ c
R ⟶ ε
R ⟶ R R
R ⟶ R | R
R ⟶ R *
```

CFGs:
```
G ⟶
G ⟶ P ; G
P ⟶ n -> W
W ⟶
W ⟶ n W
W ⟶ t W
```


Context-free grammars and languages

- What is the relationship between context-free grammars and languages (i.e., sets of strings)?
- A **derivation** is a sequence of strings:

  $s_1 \longrightarrow s_2 \longrightarrow s_3 \longrightarrow s_4 \longrightarrow s_5 \longrightarrow s_6 \longrightarrow s_7 \longrightarrow \ldots$

  in which each string s_{i+1} is obtained from the previous string s_i by choosing a production n $\longrightarrow$ w and replacing an occurrence of n in s_i with w

- In this case, we say that s_1 **derives** s_i for each i = 1, 2, ...
- We say that a context-free grammar G = (T, N, P, S) **generates** the language that contains all strings in T* that can be derived from S

Brackets is a "context-free language"

- Any language that is generated from a context-free grammar is said to be a **context-free language**
- Sample derivations for Brackets:

  B ⟶

  B ⟶ [B] ⟶ [ ]

  B ⟶ [B] ⟶ [[B]] ⟶ [[ ]]

  ...

  B ⟶ [B] ⟶ [[B]] ⟶ [[[B]]] ⟶ [[[[B]]]] ⟶ [[[[ ]]]]

  ...

Derivations and parse trees

A language of arithmetic expressions

- Many computer languages are naturally described as context-free languages (i.e., using a context-free grammar)
- Example: a simple language of expressions:

  E ⟶ n (**n** is an integer literal)

  E ⟶ E + E

  E ⟶ E * E

  E ⟶ (E)

- Terminology:

  E is a nonterminal

  +, *****, **(**, **)**, and **n** are terminals (i.e., tokens)

Deriving expressions

E ⟶ n
E ⟶ E + E
E ⟶ E * E
E ⟶ (E)

For example, 1+(2*3) is an expression:

E	⟶	E + E	E	⟶	E + E
	⟶	E + (E)		⟶	1 + E
	⟶	E + (E * E)		⟶	1 + (E)
	⟶	E + (E * 3)		⟶	1 + (E * E)
	⟶	E + (2 * 3)		⟶	1 + (2 * E)
	⟶	1 + (2 * 3)		⟶	1 + (2 * 3)

one expression, multiple derivations

Multiple derivations

- In a right-most derivation, we replace the right-most nonterminal at each step
- In a left-most derivation, we replace the left-most nonterminal at each step
- There are other choices between these extremes
- Does it matter which derivation we use?

Deriving expressions

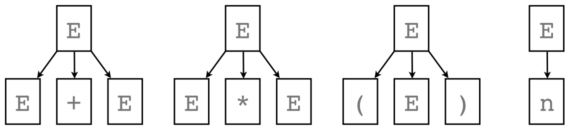
For example, 1 + 2 * 3 is an expression (no parentheses):

E	⟶	E + E	E	⟶	E * E
	⟶	E + E * E		⟶	E + E * E
	⟶	E + E * 3		⟶	1 + E * E
	⟶	E + 2 * 3		⟶	1 + 2 * E
	⟶	1 + 2 * 3		⟶	1 + 2 * 3

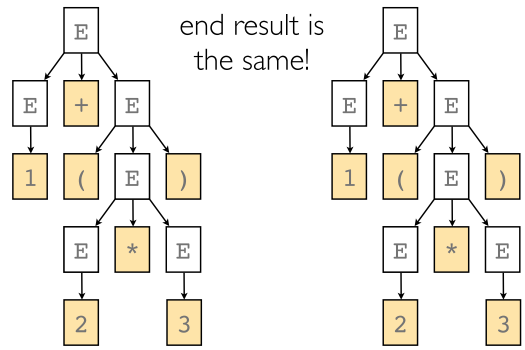
two valid derivations,
but a fundamental difference this time

Productions in graphical form

To understand the essential structure of a derivation, we will use a graphical notation to represent the productions in a grammar:

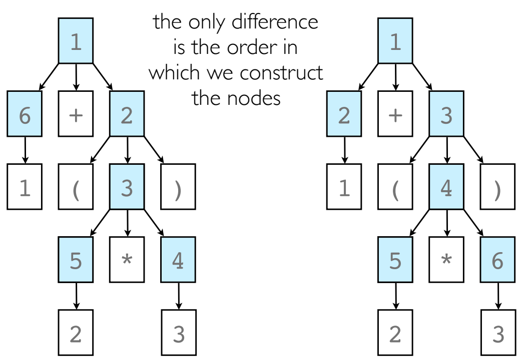


1+(2*3) revisited



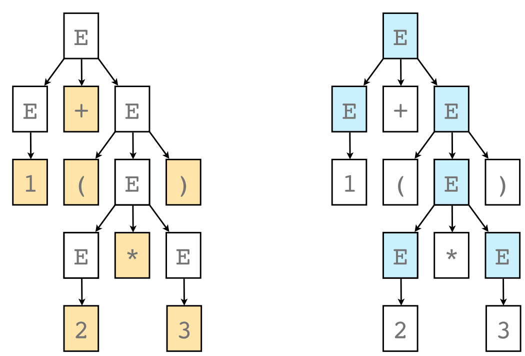
right-most left-most

Right-most vs left-most



right-most left-most

Parse trees

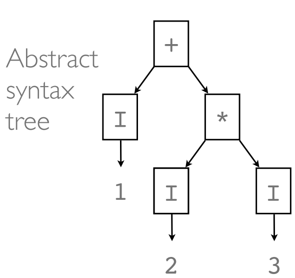


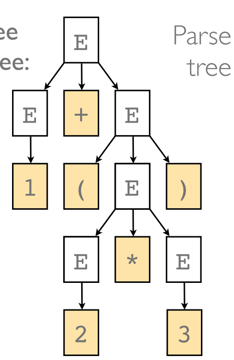
leaves are terminals, every terminal appears

interior nodes correspond to productions

Parse trees vs abstract syntax trees

Because it describes a full derivation and all of the tokens in the input, there is usually more information in a parse tree than we need for an abstract syntax tree:

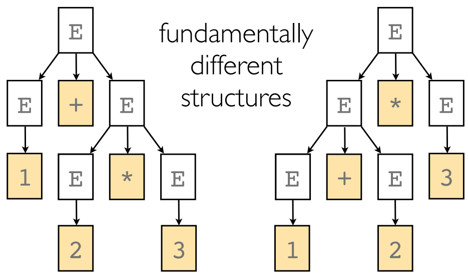




CFGs and parse trees

- Context-free grammars don't just define languages (i.e., they don't just derive *sets of strings*) ...
- ... they actually define *sets of trees*!
- The strings in the corresponding context-free language can be recovered from the leaf nodes of each tree
- Parsing works in reverse: start with a string and try to construct the original tree
- What happens in there are <u>distinct</u> tree structures with the <u>same</u> sequence of symbols on their leaves?

1+2*3 revisited



Ambiguity

- A grammar is **ambiguous** if there is a string in the corresponding language with more than one parse tree
- Example: Our grammar for expressions is ambiguous because the string "1+2*3" has two distinct parse trees

  (We could find plenty of other examples of the same problem in this grammar, but finding just one is enough to demonstrate ambiguity)
- Ambiguity is a property of a grammar, NOT a language

  We can have multiple grammars describing the same language, some ambiguous and some unambiguous

Dealing with ambiguity

- Does it matter?
- If any parse tree is as good as any other (i.e., they all have the same meaning), then just take whichever tree we get

  Example: for regular expressions, $r_1(r_2r_3)$ matches exactly the same set of strings as $(r_1r_2)r_3$, so we can parse $r_1r_2r_3$ either way
- If different trees have different meanings, then we need to choose between them:

  Disambiguating rules (e.g., operator precedence)

  Rewrite the grammar to avoid ambiguity

Precedence and grouping (fixity)

- If ⊗ has **higher precedence** than ⊕, then a ⊗ b ⊕ c should be parsed in the same way as (a ⊗ b) ⊕ c
- If ⊗ **groups/associates** to the left, then a ⊗ b ⊗ c should be parsed in the same way as (a ⊗ b) ⊗ c
- If ⊗ **groups/associates** to the right, then a ⊗ b ⊗ c should be parsed in the same way as a ⊗ (b ⊗ c)
- If ⊗ is **nonassociative**, then a ⊗ b ⊗ c should be treated as an error
- The combination of an operator's precedence and grouping is known as its **fixity**

Order of operations

- There are widely used conventions for the precedence of standard arithmetic operations. (e.g., "PEMDAS")
- Parentheses first, then multiplications, then addition

  Example: (1+2)+3*4 == (1+2)+(3*4)
- But the final decision about what fixity each operator should have rests with the language designer
- Let's suppose we want to resolve our ambiguities using:

  * has higher precedence than +

  * and + both group to the left

An unambiguous grammar for expressions

Here is an unambiguous grammar for our language of expressions:

E	⟶	E + P	expressions
E	⟶	P	
P	⟶	P * A	products
P	⟶	A	
A	⟶	(E)	atoms
A	⟶	n	(n is an integer literal)

How were these properties met?
* has higher precedence than +
* and + both group to the left

An unambiguous grammar for expressions

Here is an unambiguous grammar for our language of expressions:

E	$\longrightarrow$	E + P	expressions are sums of products
E	$\longrightarrow$	P	
P	$\longrightarrow$	P * A	products of atoms
P	$\longrightarrow$	A	
A	$\longrightarrow$	(E)	atoms
A	$\longrightarrow$	n	

An unambiguous grammar for expressions

An expression is a sum of products, each of which is a product of atoms.

Example: if a_1, ..., a_8 are atoms, then:

$$\underbrace{\underbrace{a_1 * a_2 * a_3}_{\text{product}} + \underbrace{a_4 * a_5}_{\text{product}} + \underbrace{a_6}_{\text{product}} + \underbrace{a_7 * a_8}_{\text{product}}}_{\text{expr}}$$

must be parsed as:

$$(a_1 * a_2 * a_3) + (a_4 * a_5) + (a_6) + (a_7 * a_8)$$

Multiply has been given a higher precedence than addition!

An unambiguous grammar for expressions

The right argument of + must be a product

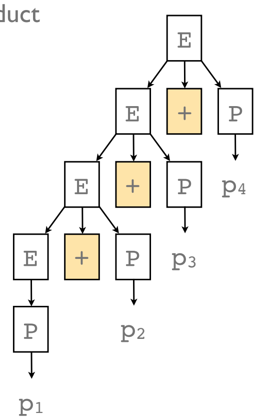
Example: if p_1, ..., p_4 are products, then:

$$\underbrace{\underbrace{\underbrace{\underbrace{p_1}_{\text{prod.}} + \underbrace{p_2}_{\text{prod.}}}_{\text{expr}} + \underbrace{p_3}_{\text{prod.}}}_{\text{expr}} + \underbrace{p_4}_{\text{prod.}}}_{\text{expr}}$$

must be parsed as:

$$(((p_1 + p_2) + p_3) + p_4)$$

In other words: + groups to the left



Controlling grouping

E $\longrightarrow$ E + P

E $\longrightarrow$ P

recursion on the left results in left grouping

P $\longrightarrow$ P * A

P $\longrightarrow$ A

similarly for *

for right grouping, change the recursion

E $\longrightarrow$ P + E

E $\longrightarrow$ P

for nonassociative operators, no recursion on either side

P $\longrightarrow$ A * A

P $\longrightarrow$ A

Fragments from the official Java grammar

increasing precedence

MultiplicativeExpression:
- *UnaryExpression*
- *MultiplicativeExpression* `*` *UnaryExpression*
- *MultiplicativeExpression* `/` *UnaryExpression*
- *MultiplicativeExpression* `%` *UnaryExpression*

AdditiveExpression:
- *MultiplicativeExpression*
- *AdditiveExpression* `+` *MultiplicativeExpression*
- *AdditiveExpression* `-` *MultiplicativeExpression*

ShiftExpression:
- *AdditiveExpression*
- *ShiftExpression* `<<` *AdditiveExpression*
- *ShiftExpression* `>>` *AdditiveExpression*
- *ShiftExpression* `>>>` *AdditiveExpression*

RelationalExpression:
- *ShiftExpression*
- *RelationalExpression* `<` *ShiftExpression*
- *RelationalExpression* `>` *ShiftExpression*
- *RelationalExpression* `<=` *ShiftExpression*
- *RelationalExpression* `>=` *ShiftExpression*
- *RelationalExpression* `instanceof` *ReferenceType*

EqualityExpression:
- *RelationalExpression*
- *EqualityExpression* `==` *RelationalExpression*
- *EqualityExpression* `!=` *RelationalExpression*

AndExpression:
- *EqualityExpression*
- *AndExpression* `&` *EqualityExpression*

ExclusiveOrExpression:
- *AndExpression*
- *ExclusiveOrExpression* `^` *AndExpression*

InclusiveOrExpression:
- *ExclusiveOrExpression*
- *InclusiveOrExpression* `|` *ExclusiveOrExpression*

ConditionalAndExpression:
- *InclusiveOrExpression*
- *ConditionalAndExpression* `&&` *InclusiveOrExpression*

ConditionalOrExpression:
- *ConditionalAndExpression*
- *ConditionalOrExpression* `||` *ConditionalAndExpression*

ConditionalExpression:
- *ConditionalOrExpression*
- *ConditionalOrExpression* `?` Expression `:` *ConditionalExpression*

decreasing precedence

Context free languages summary

- Context free grammars can be used to describe a significantly larger family of languages than regular expressions
  - including many of the languages we encounter in practice
- Parse trees are graphical descriptions of derivations that
  - can reflect the grammatical structure of the input
  - can highlight ambiguities in the grammar
  - include more detail than is typically needed for an AST
- Grammars can be written so that operators are treated as having different precedence and grouping behaviors
  - This can help to avoid ambiguity