MIT 6.035
Specifying Languages with Regular Expressions and Context-Free Grammars

Martin Rinard
Laboratory for Computer Science
Massachusetts Institute of Technology

Language Definition Problem

- How to precisely define language
- Layered structure of language definition
  - Start with a set of letters in language
  - Lexical structure - identifies "words" in language (each word is a sequence of letters)
  - Syntactic structure - identifies "sentences" in language (each sentence is a sequence of words)
  - Semantics - meaning of program (specifies what result should be for each input)
  - Today's topic: lexical and syntactic structures

Specifying Formal Languages

- Huge Triumph of Computer Science
  - Beautiful Theoretical Results
  - Practical Techniques and Applications
- Two Dual Notions
  - Generative approach
    (grammar or regular expression)
  - Recognition approach (automaton)
- Lots of theorems about converting one approach automatically to another

Specifying Lexical Structure Using Regular Expressions

- Have some alphabet $\sum$ = set of letters
- Regular expressions are built from:
  - ε - empty string
  - Any letter from alphabet $\sum$
  - $r_1 r_2$ – regular expression r_1 followed by r_2 (sequence)
  - $r_1 | r_2$ – either regular expression r_1 or r_2 (choice)
  - r^* - iterated sequence and choice $\varepsilon \mid r \mid rr \mid \ldots$
  - Parentheses to indicate grouping/precedence

Concept of Regular Expression Generating a String

Rewrite regular expression until have only a sequence of letters (string) left

General Rules

1) $r_1 | r_2 \rightarrow r_1$

2) $r_1 | r_2 \rightarrow r_2$

3) $r^* \rightarrow rr^*$

4) $r^* \rightarrow \varepsilon$

Example

(0 | 1)*.(0|1)*

(0 | 1)(0 | 1)*.(0|1)*

1(0|1)*.(0|1)*

1.(0|1)*

1.(0|1)(0|1)*

1.(0|1)

1.0

Nondeterminism in Generation

- Rewriting is similar to equational reasoning
- But different rule applications may yield different final results

Example 1	Example 2
(0|1)*.(0|1)*	(0|1)*.(0|1)*
(0|1)(0|1)*.(0|1)*	(0|1)(0|1)*.(0|1)*
1(0|1)*.(0|1)*	0(0|1)*.(0|1)*
1.(0|1)*	0.(0|1)*
1.(0|1)(0|1)*	0.(0|1)(0|1)*
1.(0|1)	0.(0|1)
1.0	0.1

Concept of Language Generated by Regular Expressions

- Set of all strings generated by a regular expression is language of regular expression
- In general, language may be (countably) infinite
- String in language is often called a token

Examples of Languages and Regular Expressions

- $\sum$ = { 0, 1, . }
  - (0|1)*.(0|1)* - Binary floating point numbers
  - (00)* - even-length all-zero strings
  - 1*(01*01*)* - strings with even number of zeros
- $\sum$ = { a,b,c, 0, 1, 2 }
  - (a|b|c)(a|b|c|0|1|2)* - alphanumeric identifiers
  - (0|1|2)* - trinary numbers

Alternate Abstraction
Finite-State Automata

- Alphabet Σ
- Set of states with initial and accept states
- Transitions between states, labeled with letters

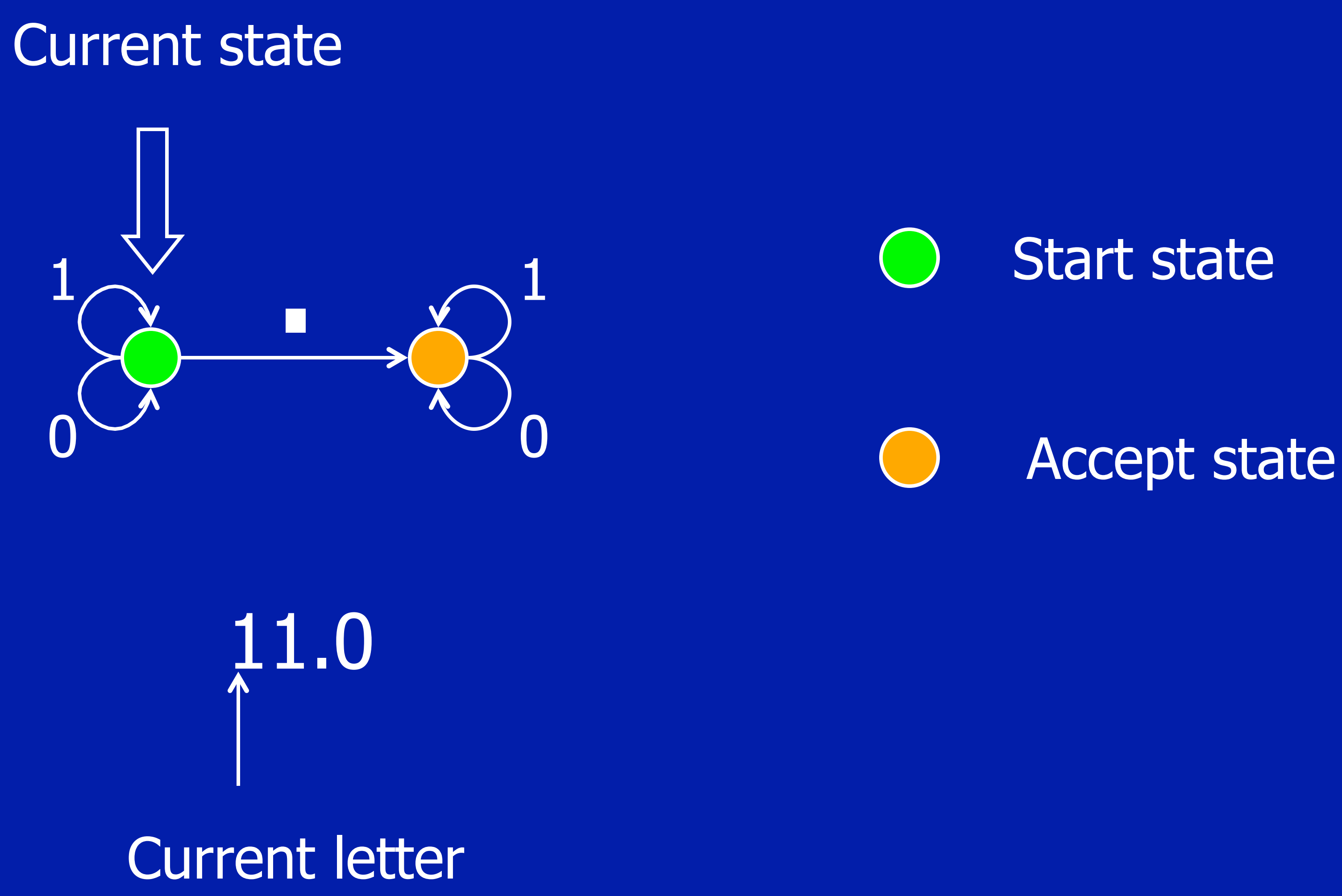
(0|1)*.(0|1)*

Automaton Accepting String

Conceptually, run string through automaton

- Have current state and current letter in string
- Start with start state and first letter in string
- At each step, match current letter against a transition whose label is same as letter
- Continue until reach end of string or match fails
- If end in accept state, automaton accepts string
- Language of automaton is set of strings it accepts

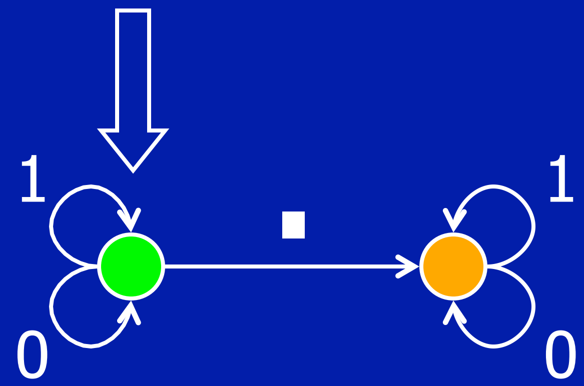
Example



Current state
1
.
1
0
0
11.0
Current letter
Start state
Accept state

Example

Current state



Start state

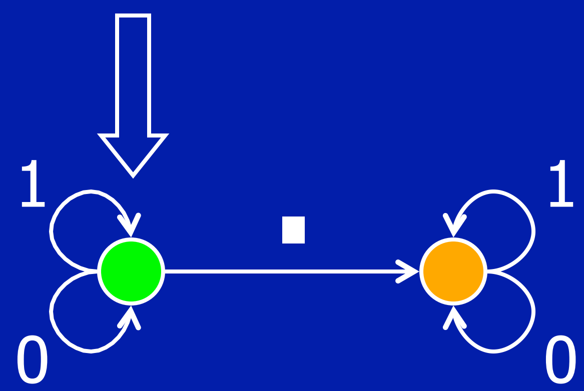
Accept state

11.0

Current letter

Example

Current state



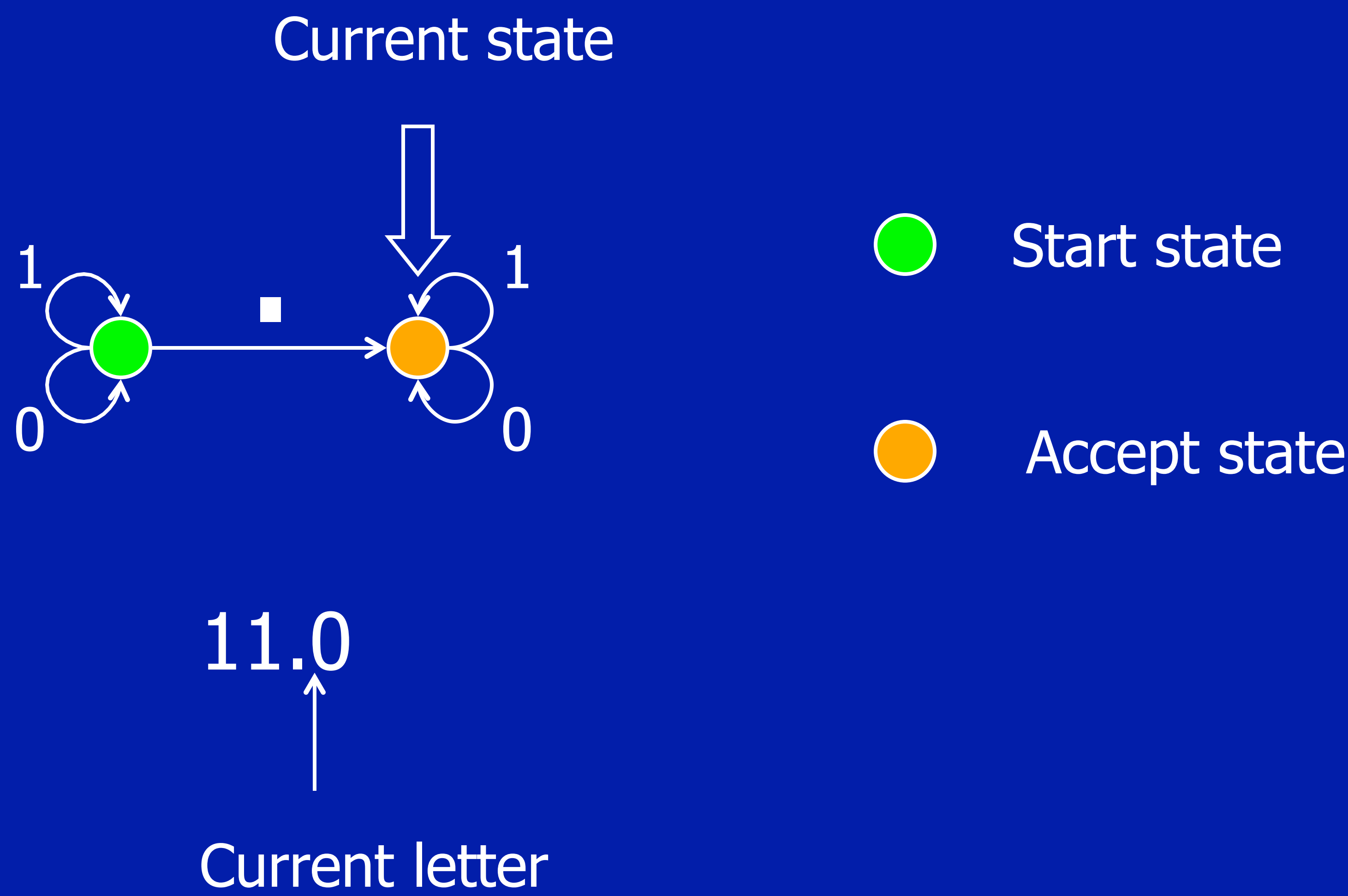
Start state

Accept state

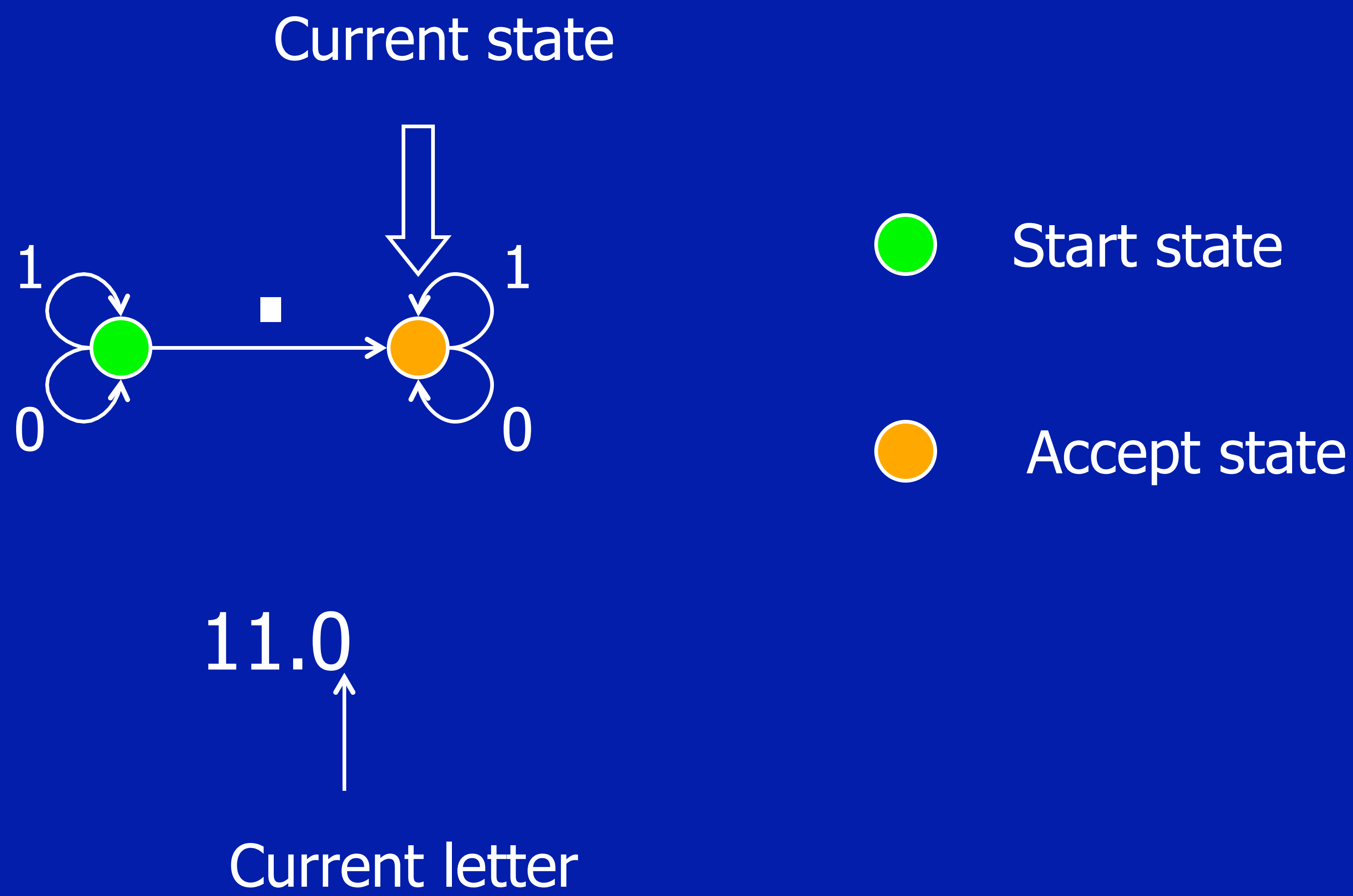
11.0

Current letter

Example

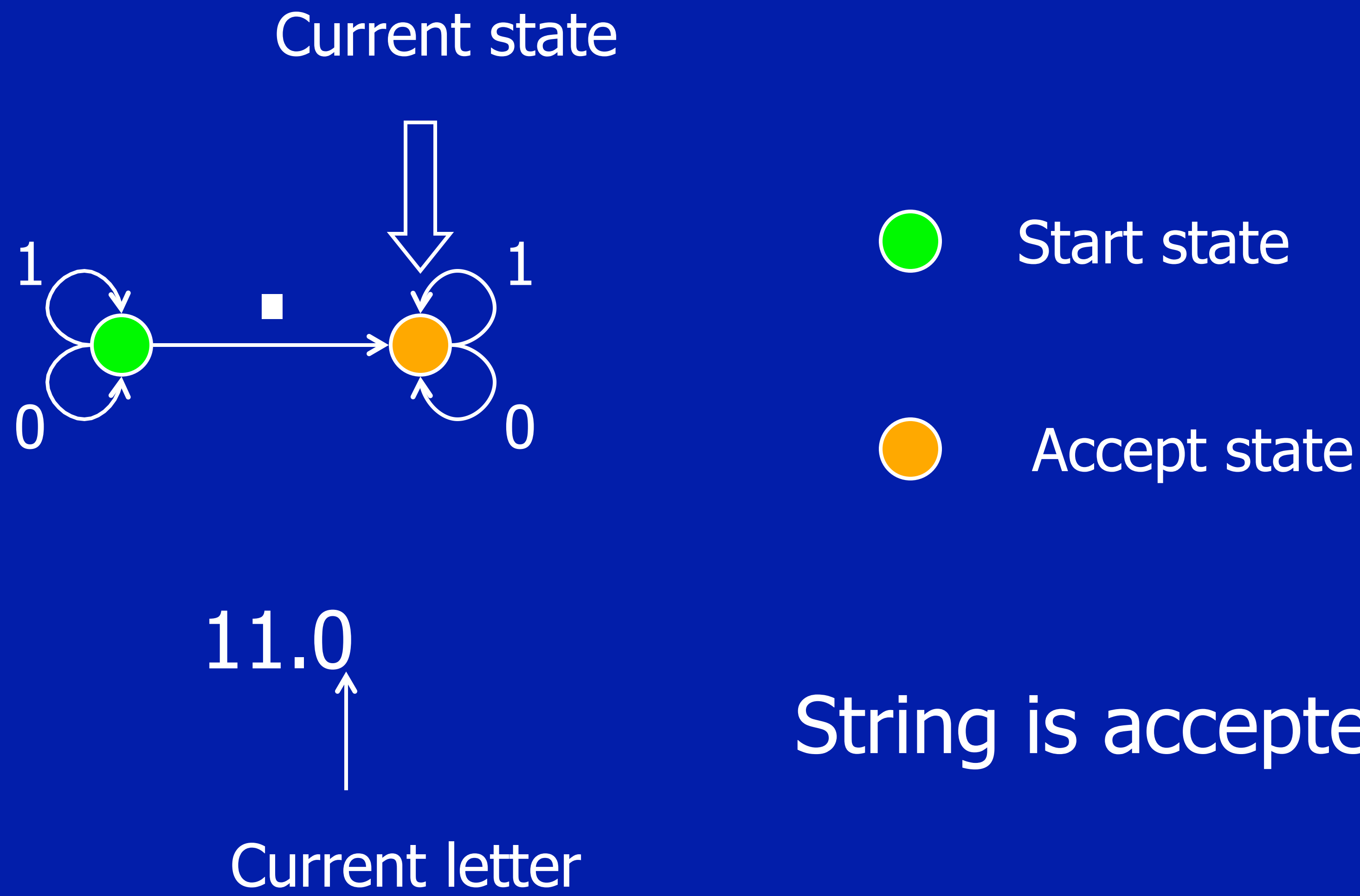

Current state
1
0
.
1
0
Start state
Accept state
11.0
Current letter

Example



Current state
1
.
1
0
0
Start state
Accept state
11.0
Current letter

Example



String is accepted!

Generative Versus Recognition

- Regular expressions give you a way to generate all strings in language
- Automata give you a way to recognize if a specific string is in language
  - Philosophically very different
  - Theoretically equivalent (for regular expressions and automata)
- Standard approach
  - Use regular expressions when define language
  - Translated automatically into automata for implementation

From Regular Expressions to Automata

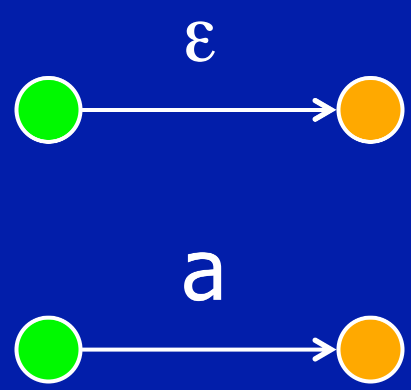
- Construction by structural induction
- Given an arbitrary regular expression r
- Assume we can convert r to an automaton with
  - One start state
  - One accept state
- Show how to convert all constructors to deliver an automaton with
  - One start state
  - One accept state

Basic Constructs

- Start state
- Accept state

ε

$a \in \Sigma$

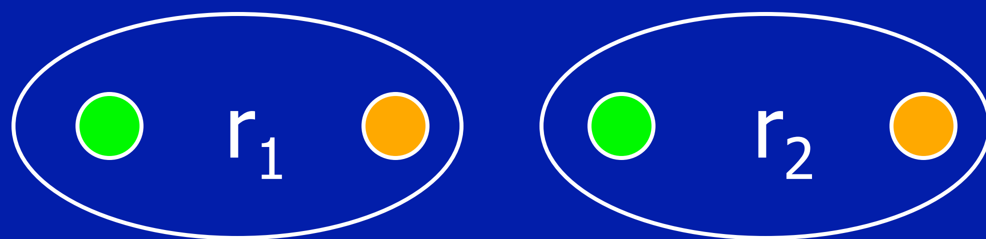


Sequence

Start state

Accept state

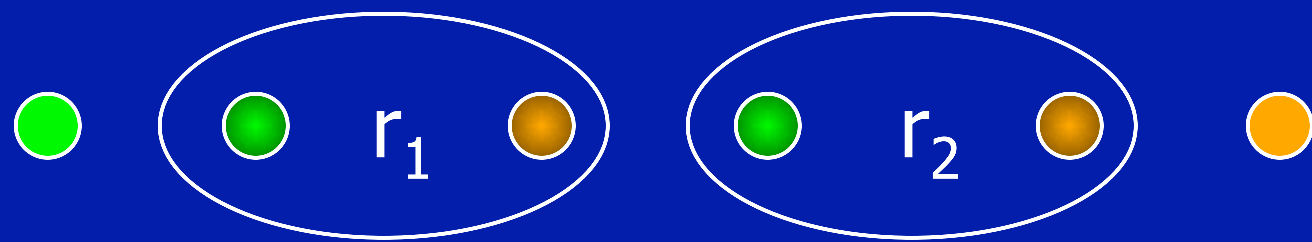
$r_1 r_2$



Sequence

- Old start state
- Old accept state
- Start state
- Accept state

$r_1 r_2$



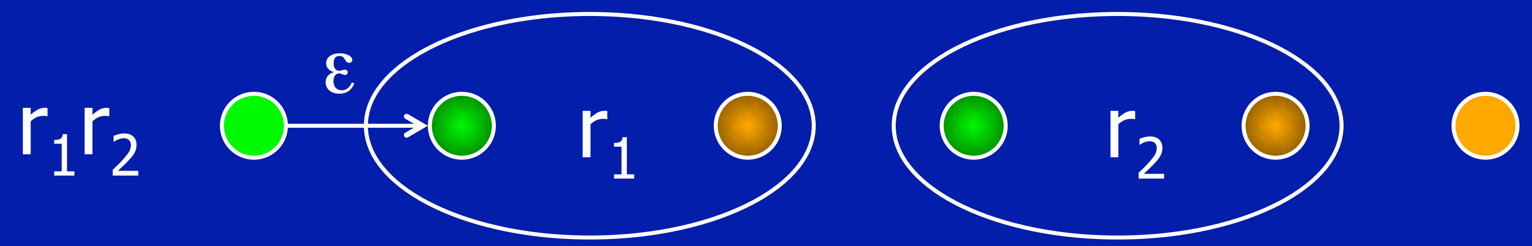
Sequence

Old start state

Start state

Old accept state

Accept state



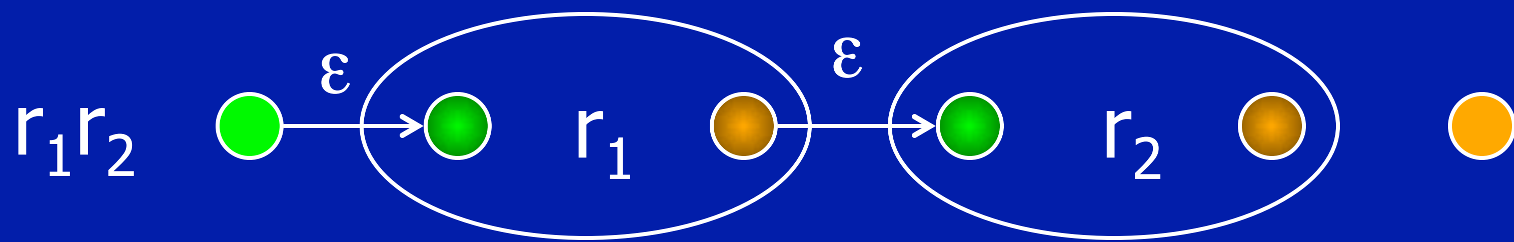
Sequence

Old start state

Old accept state

Start state

Accept state



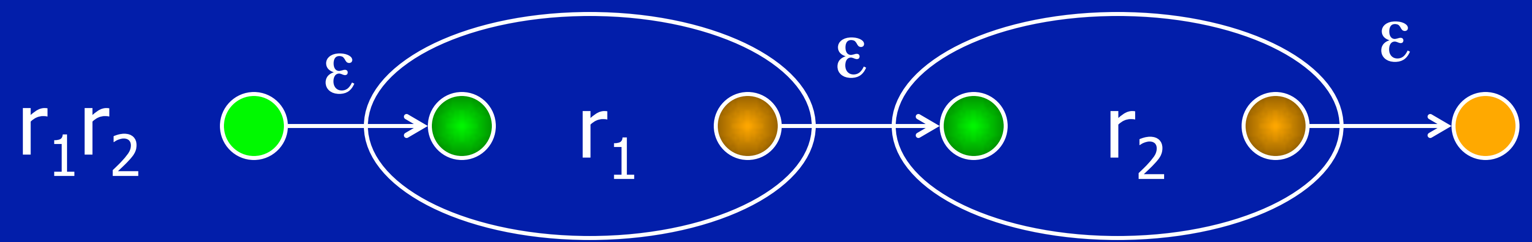
Sequence

Old start state

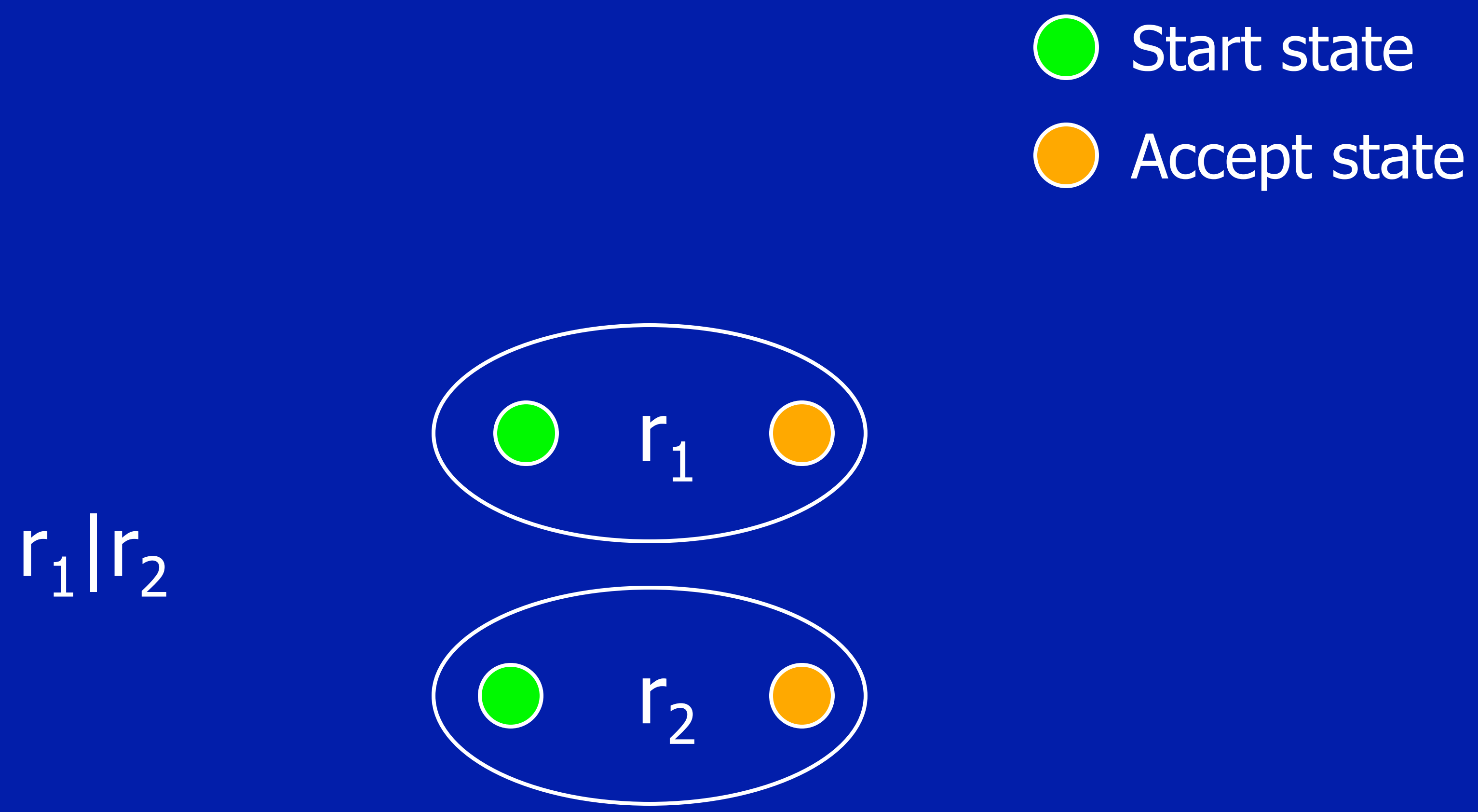
Old accept state

Start state

Accept state



Choice



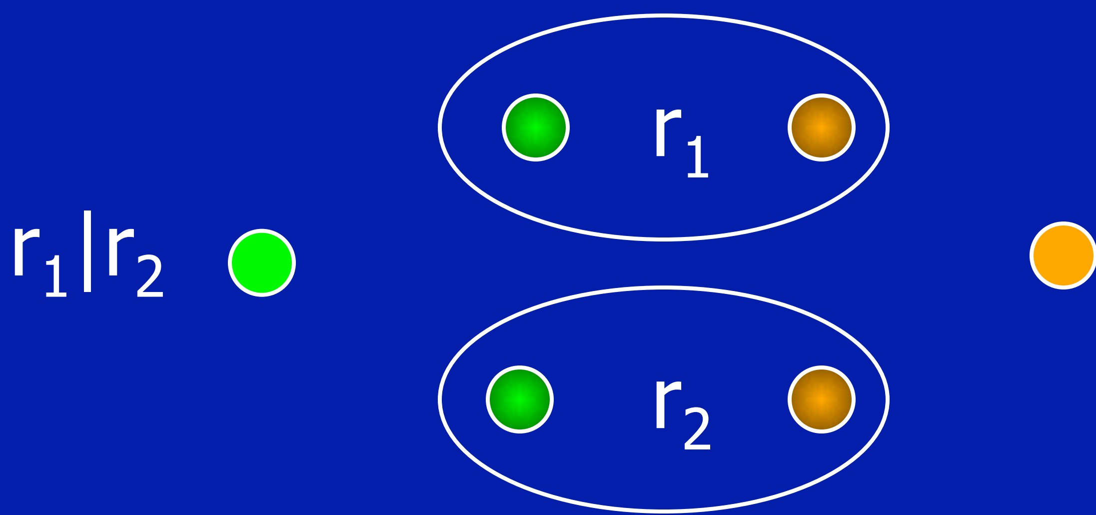
Choice

Old start state

Old accept state

Start state

Accept state



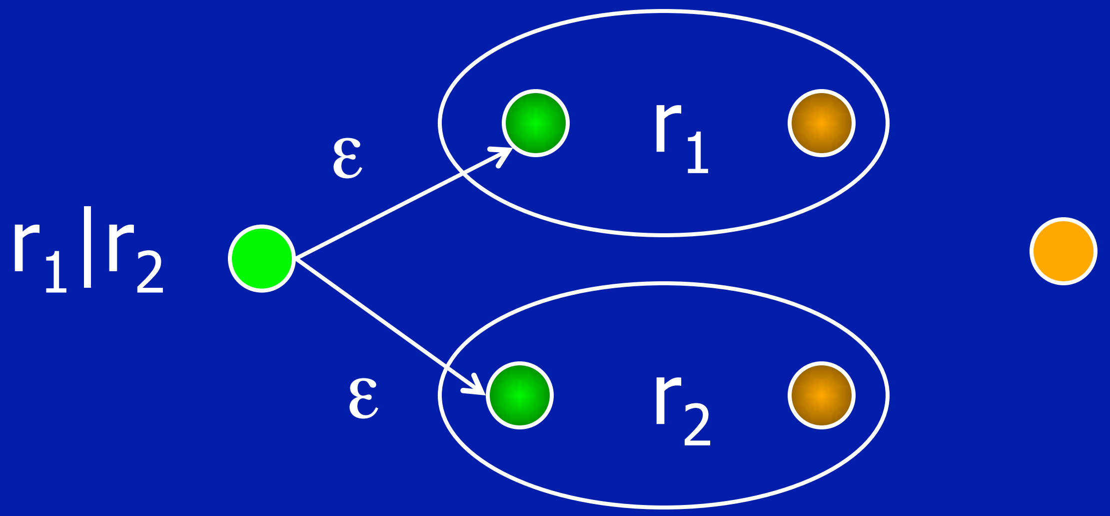
Choice

Old start state

Old accept state

Start state

Accept state



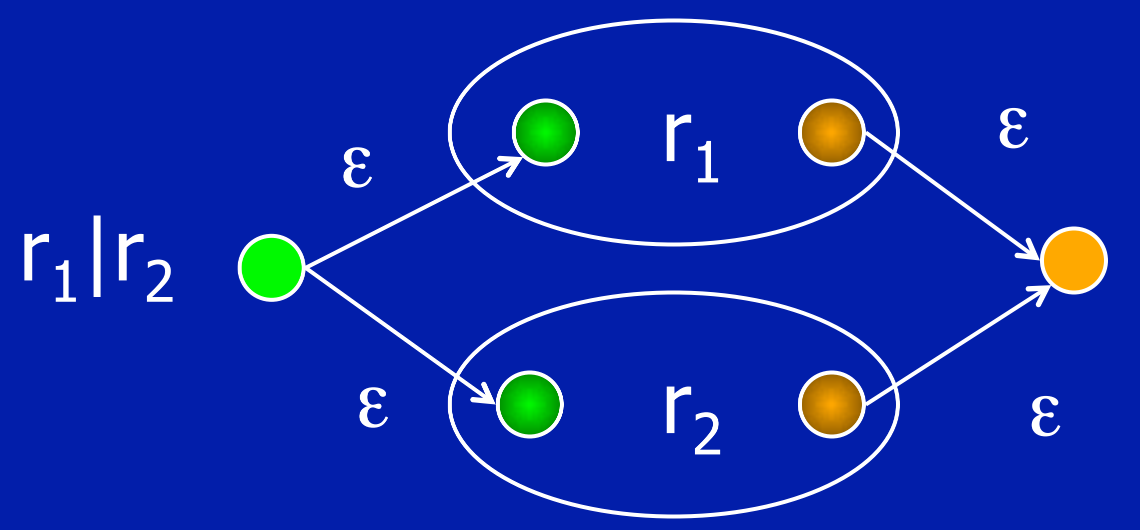
Choice

Old start state

Start state

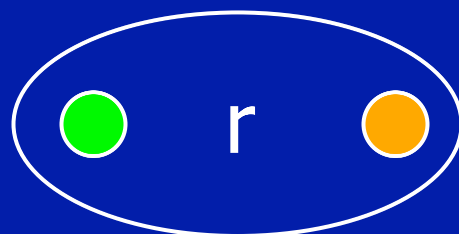
Old accept state

Accept state



Kleene Star

r*



Kleene Star

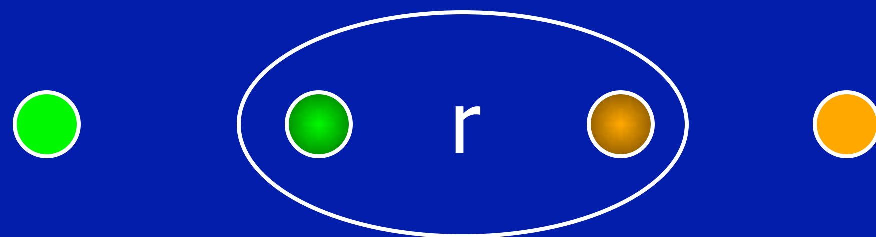
Old start state

Start state

Old accept state

Accept state

r*



Kleene Star

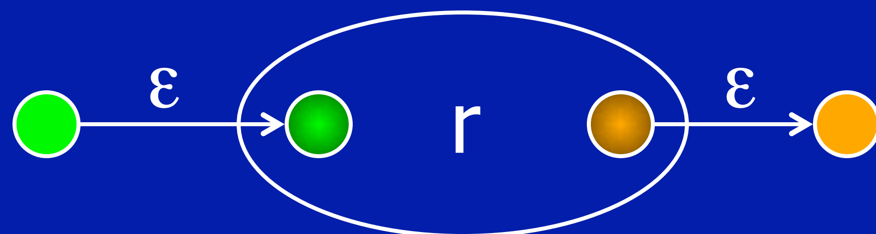
Old start state

Old accept state

Start state

Accept state

r*



Kleene Star

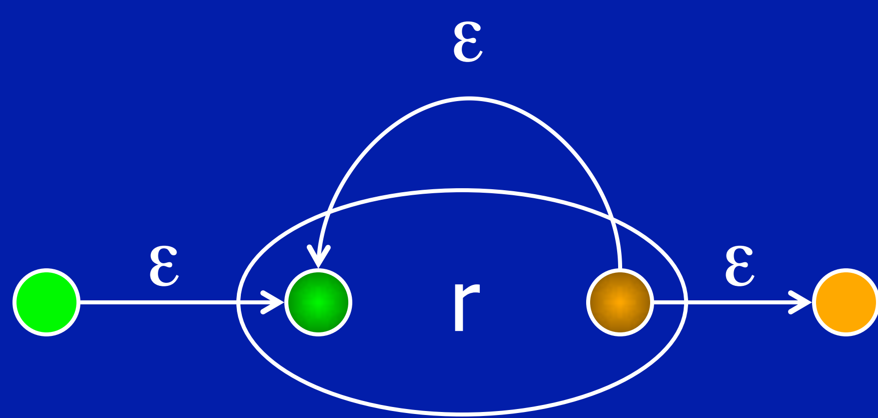
Old start state

Old accept state

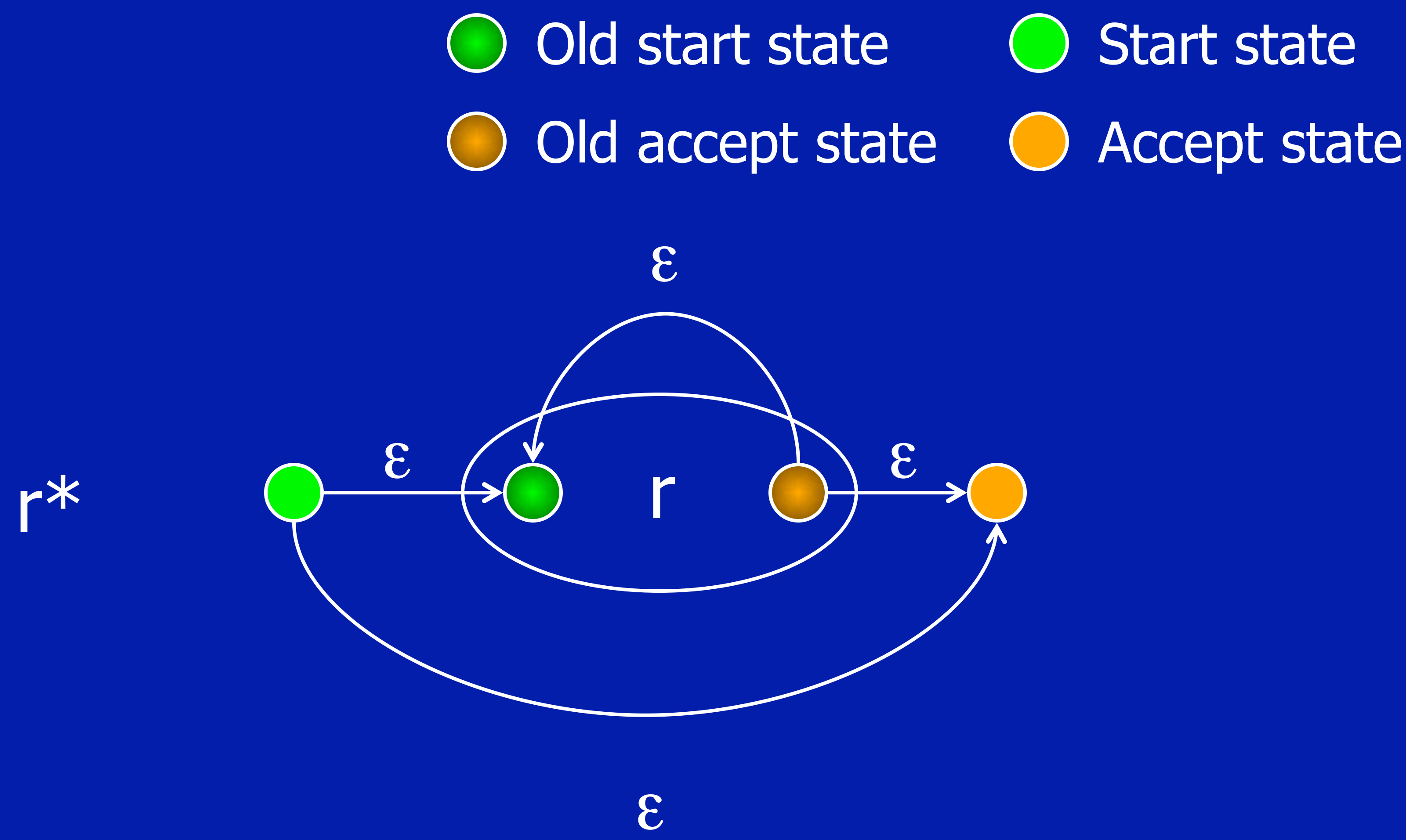
Start state

Accept state

r*



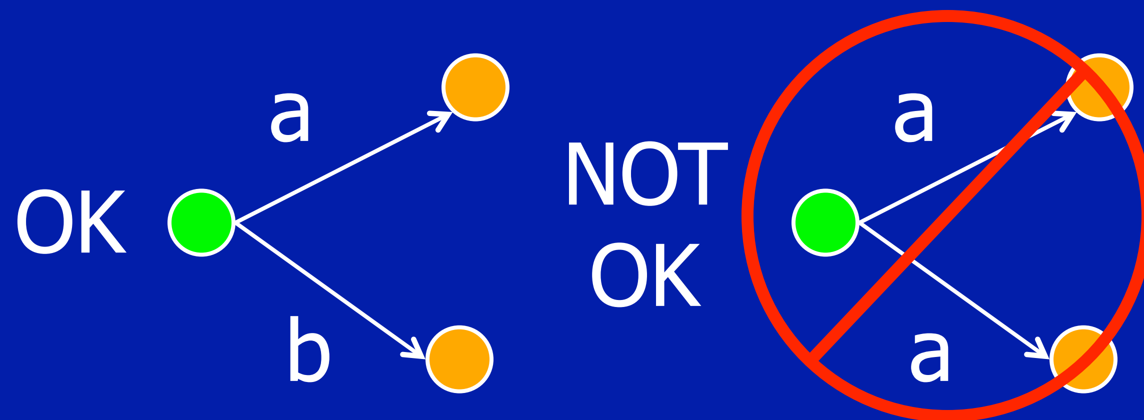
Kleene Star



Old start state
Start state
Old accept state
Accept state
ε
r*
ε
r
ε
ε

NFA vs. DFA

- DFA
  - No ε transitions
  - At most one transition from each state for each letter



- NFA – neither restriction

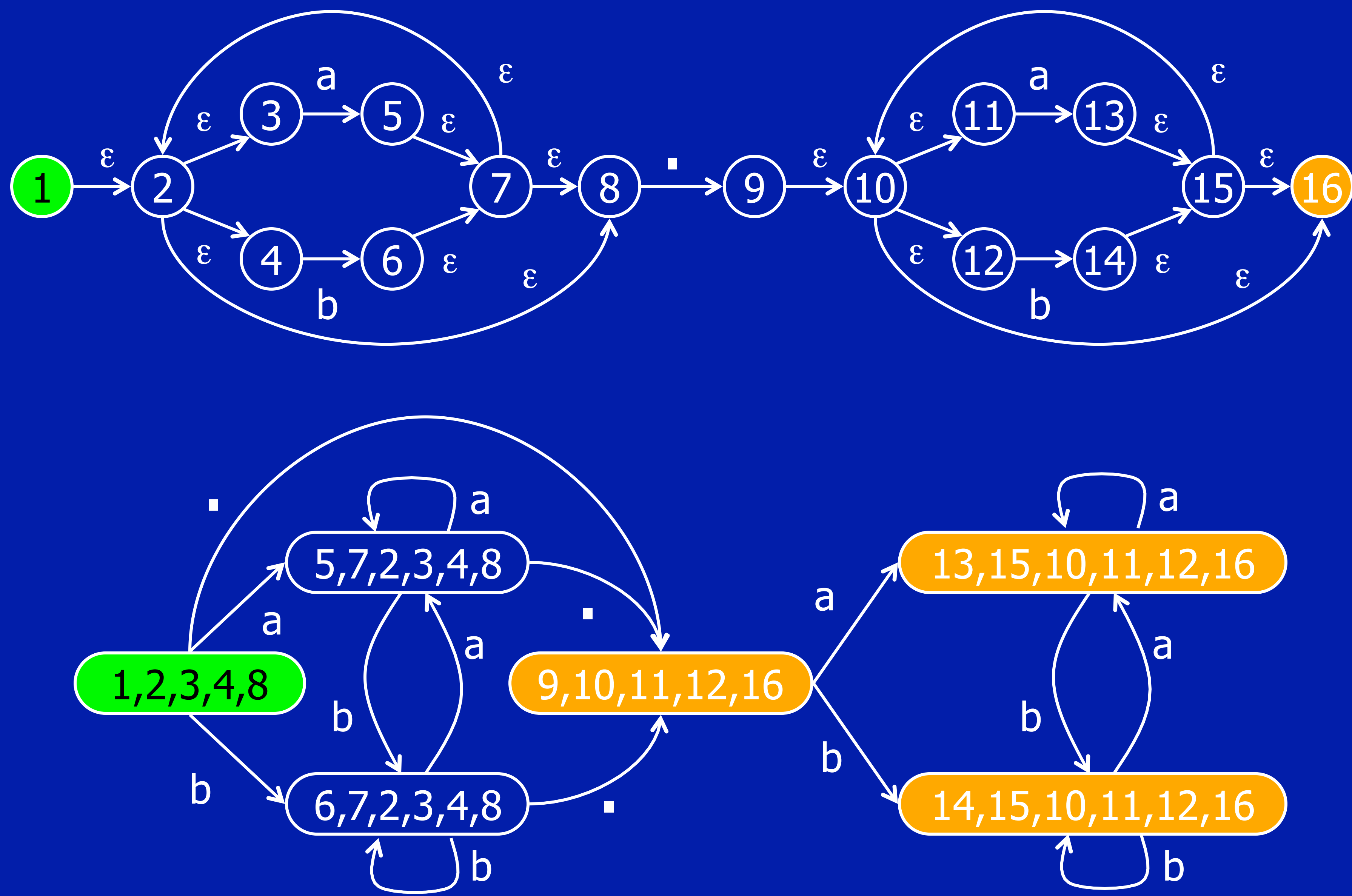
Conversions

- Our regular expression to automata conversion produces an NFA
- Would like to have a DFA to make recognition algorithm simpler
- Can convert from NFA to DFA (but DFA may be exponentially larger than NFA)

NFA to DFA Construction

- DFA has a state for each subset of states in NFA
  - DFA start state corresponds to set of states reachable by following ε transitions from NFA start state
  - DFA state is an accept state if an NFA accept state is in its set of NFA states
- To compute the transition for a given DFA state D and letter a
  - Set S to empty set
  - Find the set N of D's NFA states
    - For all NFA states n in N
      - Compute set of states N' that the NFA may be in after matching a
      - Set S to S union N'
  - If S is nonempty, there is a transition for a from D to the DFA state that has the set S of NFA states
  - Otherwise, there is no transition for a from D

NFA to DFA Example for (a|b)*.(a|b)*



Lexical Structure in Languages

Each language typically has several categories of words. In a typical programming language:

- Keywords (if, while)
- Arithmetic Operations (+, -, *, /)
- Integer numbers (1, 2, 45, 67)
- Floating point numbers (1.0, .2, 3.337)
- Identifiers (abc, i, j, ab345)

- Typically have a lexical category for each keyword and/or each category
- Each lexical category defined by regexp

Lexical Categories Example

- IfKeyword = if
- WhileKeyword = while
- Operator = +|-|*|/
- Integer = [0-9] [0-9]*
- Float = [0-9]*. [0-9]*
- Identifier = [a-z]([a-z]|[0-9])*
- Note that [0-9] = (0|1|2|3|4|5|6|7|8|9)
  [a-z] = (a|b|c|…|y|z)
- Will use lexical categories in next level

Programming Language Syntax

- Regular languages suboptimal for specifying programming language syntax
- Why? Constructs with nested syntax
  - (a+(b-c))*(d-(x-(y-z)))
  - if (x < y) if (y < z) a = 5 else a = 6 else a = 7
- Regular languages lack state required to model nesting
- Canonical example: nested expressions
- No regular expression for language of parenthesized expressions

Solution – Context-Free Grammar

- Set of terminals

  { Op, Int, Open, Close }

  Each terminal defined

  by regular expression
- Set of nonterminals

  { *Start*, *Expr* }
- Set of productions
  - Single nonterminal on LHS
  - Sequence of terminals and nonterminals on RHS

Op = +|-|*|/

Int = [0-9] [0-9]*

Open = <

Close = >

Start → *Expr*

Expr → *Expr* Op *Expr*

Expr → Int

Expr → Open *Expr* Close

Production Game

have a current string

start with *Start* nonterminal

loop until no more nonterminals

- choose a nonterminal in current string
- choose a production with nonterminal in LHS
- replace nonterminal with RHS of production

substitute regular expressions with corresponding strings

generated string is in language

Note: different choices produce different strings

Sample Derivation

Op = +|-|*|/

Int = [0-9] [0-9]*

Open = <

Close = >

1) *Start* → *Expr*
2) *Expr* → *Expr* Op *Expr*
3) *Expr* → Int
4) *Expr* → Open *Expr* Close

Start

Expr

Expr Op *Expr*

Open *Expr* Close Op *Expr*

Open *Expr* Op *Expr* Close Op *Expr*

Open Int Op *Expr* Close Op *Expr*

Open Int Op *Expr* Close Op Int

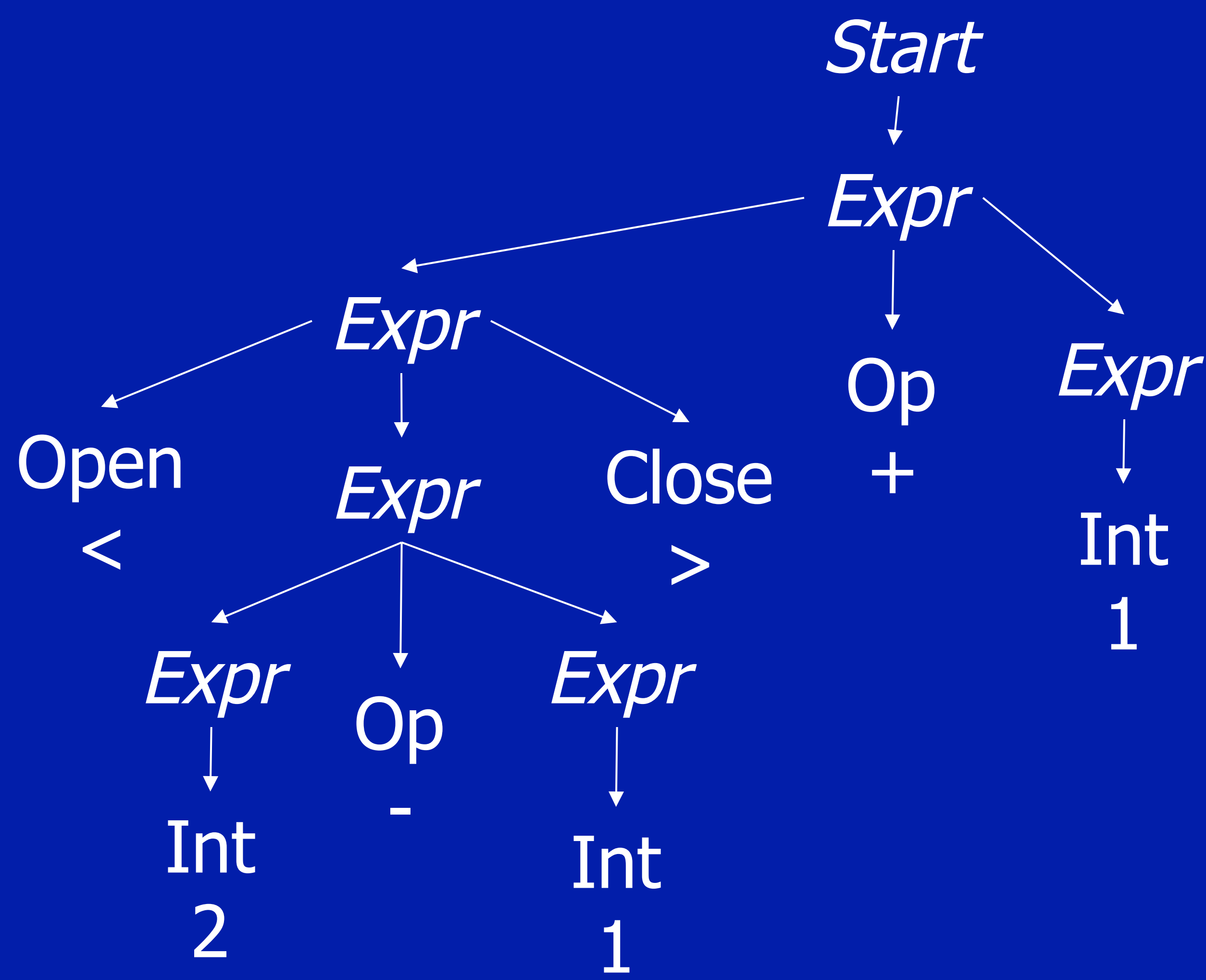
Open Int Op Int Close Op Int

< 2 - 1 > + 1

Parse Tree

- Internal Nodes: Nonterminals
- Leaves: Terminals
- Edges:
  - From Nonterminal of LHS of production
  - To Nodes from RHS of production
- Captures derivation of string

Parse Tree for <2-1>+1



Ambiguity in Grammar

Grammar is ambiguous if there are multiple derivations (therefore multiple parse trees) for a single string

Derivation and parse tree usually reflect semantics of the program

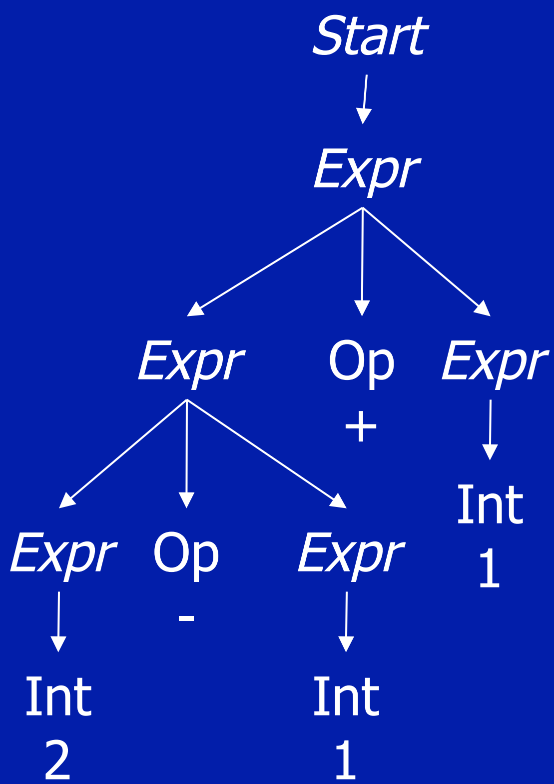
Ambiguity in grammar often reflects ambiguity in semantics of language

(which is considered undesirable)

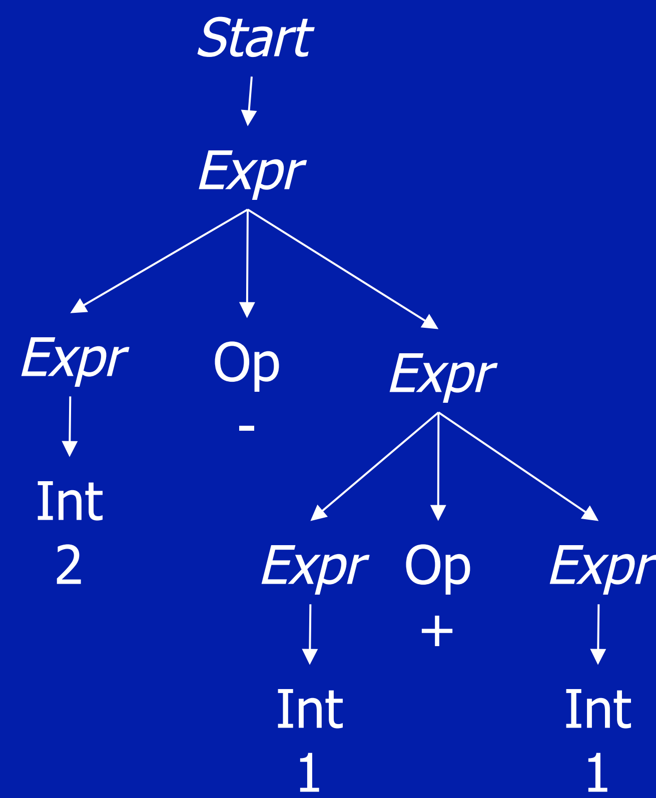
Ambiguity Example

Two parse trees for 2-1+1

Tree corresponding to <2-1>+1



Tree corresponding to 2-<1+1>



Eliminating Ambiguity

Solution: hack the grammar

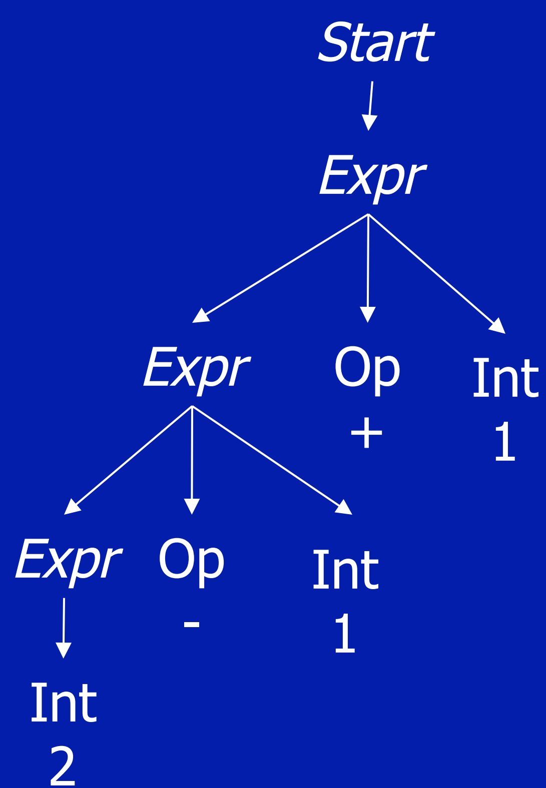
Original Grammar	Hacked Grammar
Start → *Expr*	*Start* → *Expr*
Expr → *Expr* Op *Expr*	*Expr* → *Expr* Op Int
Expr → Int	*Expr* → Int
Expr → Open *Expr* Close	*Expr* → Open *Expr* Close

Conceptually, makes all operators associate to left

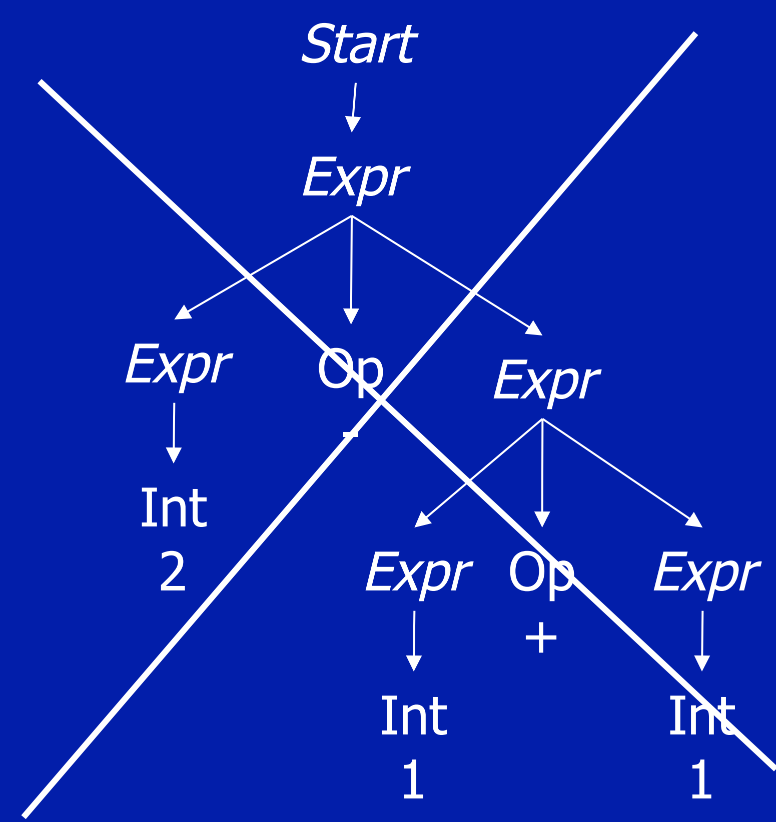
Parse Trees for Hacked Grammar

Only one parse tree for 2-1+1!

Valid parse tree



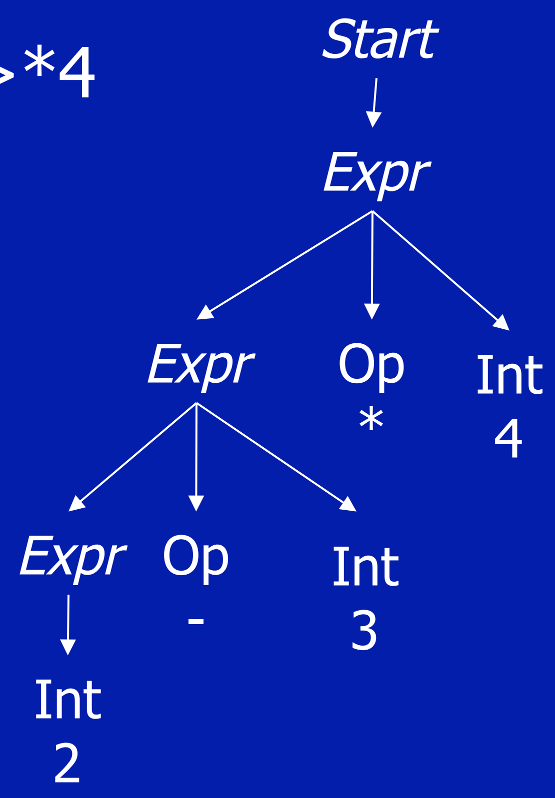
No longer valid parse tree



Precedence Violations

- All operators associate to left
- Violates precedence of * over +
  - 2-3*4 associates like <2-3>*4

Parse tree for 2-3*4



Hacking Around Precedence

Original Grammar

Op = +|-|*|/

Int = [0-9] [0-9]*

Open = <

Close = >

Start → *Expr*

Expr → *Expr* Op Int

Expr → Int

Expr → Open *Expr* Close

Hacked Grammar

AddOp = +|-

MulOp = *|/

Int = [0-9] [0-9]*

Open = <

Close = >

Start → *Expr*

Expr → *Expr* AddOp *Term*

Expr → *Term*

Term → *Term* MulOp *Num*

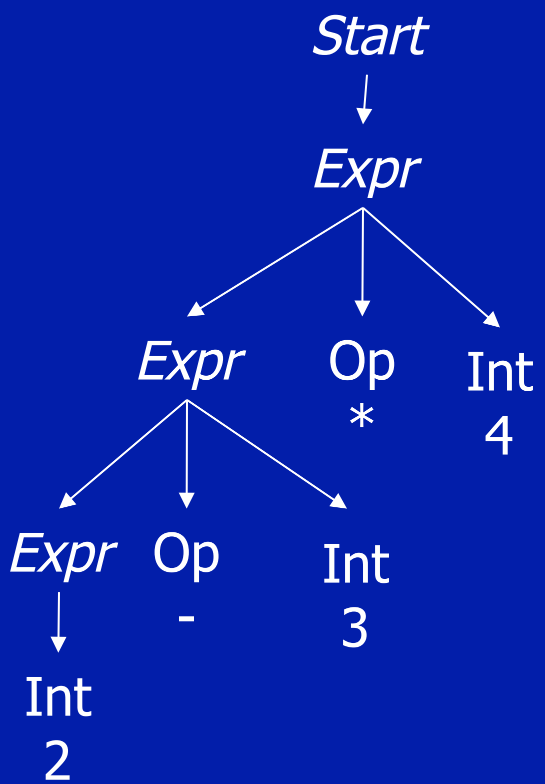
Term → *Num*

Num → Int

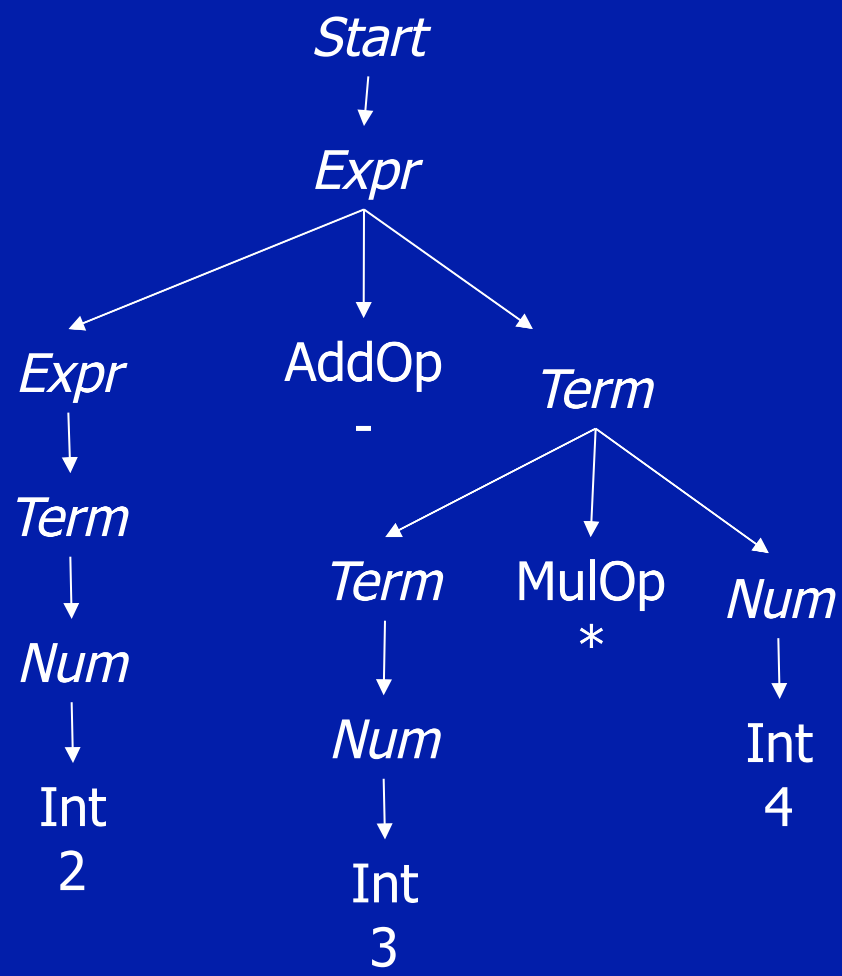
Num → Open *Expr* Close

Parse Tree Changes

Old parse tree
for 2-3*4



New parse tree
for 2-3*4



General Idea

- Group Operators into Precedence Levels
  - * and / are at top level, bind strongest
  - + and - are at next level, bind next strongest
- Nonterminal for each Precedence Level
  - *Term* is nonterminal for * and /
  - *Expr* is nonterminal for + and -
- Can make operators left or right associative within each level
- Generalizes for arbitrary levels of precedence

Parser

- Converts program into a parse tree
- Can be written by hand
- Or produced automatically by parser generator
  - Accepts a grammar as input
  - Produces a parser as output
- Practical problem
  - Parse tree for hacked grammar is complicated
  - Would like to start with more intuitive parse tree

Solution

- Abstract versus Concrete Syntax
  - Abstract syntax corresponds to “intuitive” way of thinking of structure of program
    - Omits details like superfluous keywords that are there to make the language unambiguous
    - Abstract syntax may be ambiguous
  - Concrete Syntax corresponds to full grammar used to parse the language
- Parsers are often written to produce abstract syntax trees.

Abstract Syntax Trees

- Start with intuitive but ambiguous grammar
- Hack grammar to make it unambiguous
  - Concrete parse trees
  - Less intuitive
- Convert concrete parse trees to abstract syntax trees
  - Correspond to intuitive grammar for language
  - Simpler for program to manipulate

Example

Hacked Unambiguous Grammar

AddOp = +|-

MulOp = *|/

Int = [0-9] [0-9]*

Open = <

Close = >

Start → *Expr*

Expr → *Expr* AddOp *Term*

Expr → *Term*

Term → *Term* MulOp *Num*

Term → *Num*

Num → Int

Num → Open *Expr* Close

Intuitive but Ambiguous Grammar

Op = *|/|+|-

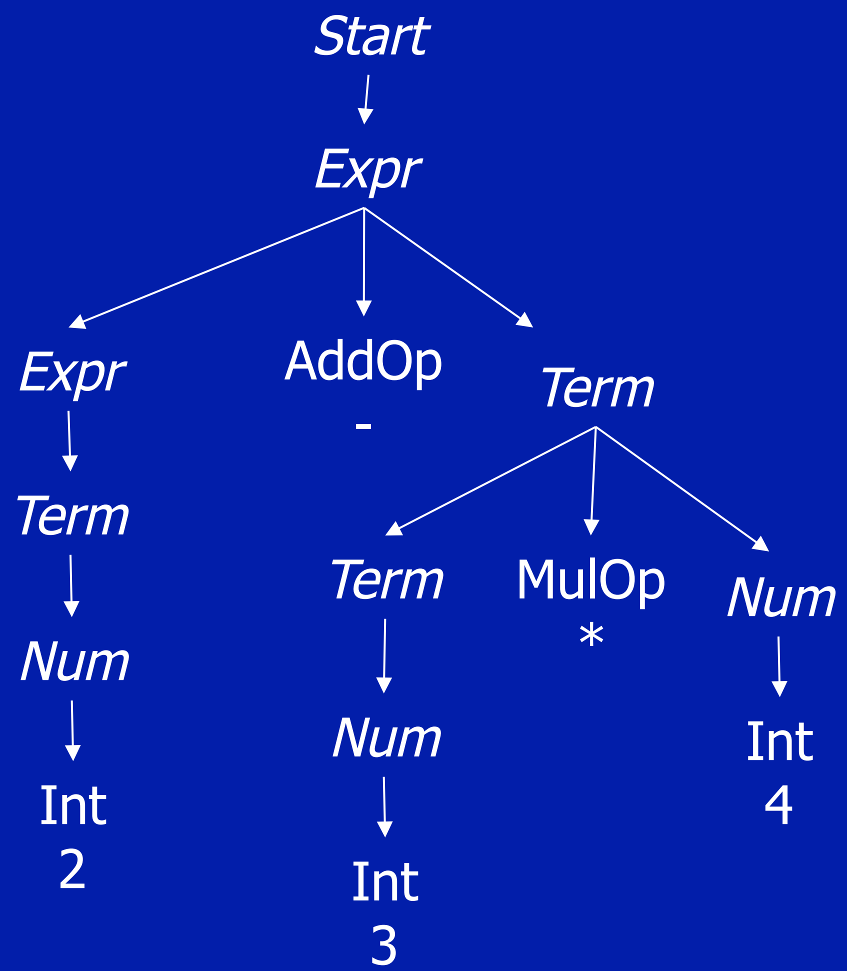
Int = [0-9] [0-9]*

Start → *Expr*

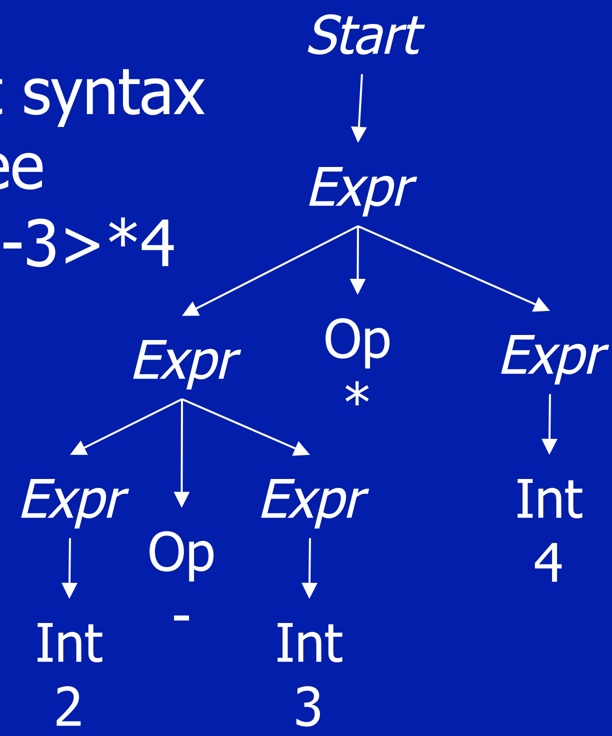
Expr → *Expr* Op *Expr*

Expr → Int

Concrete parse tree
for <2-3>*4

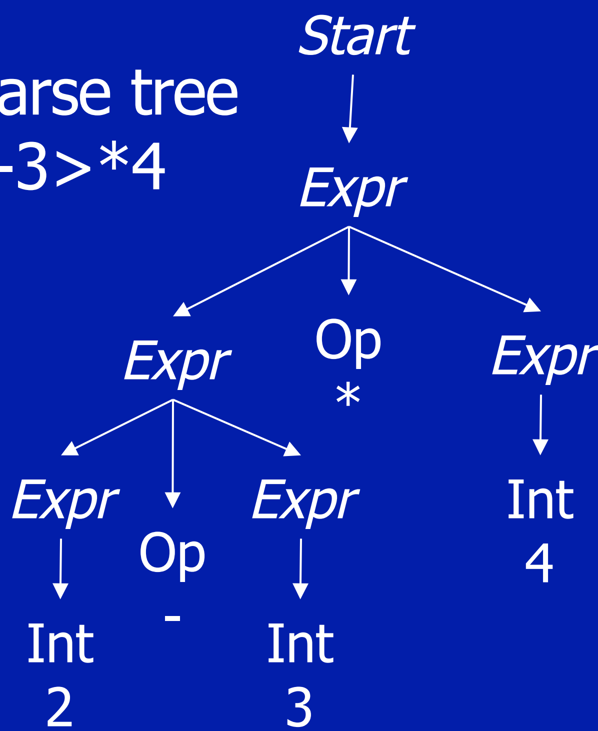


Abstract syntax tree
for <2-3>*4

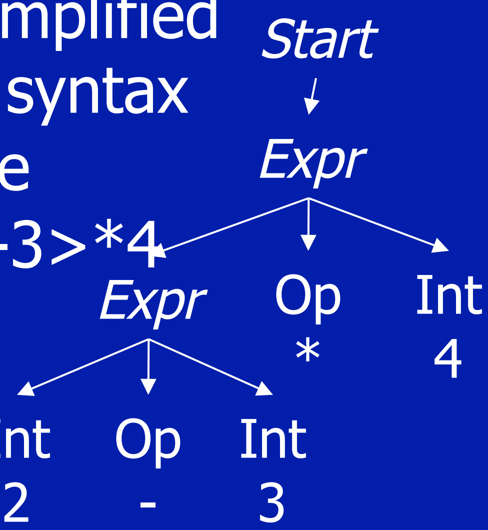


- Uses intuitive grammar
- Eliminates superfluous terminals
  - Open
  - Close

Abstract parse tree
for <2-3>*4



Further simplified
abstract syntax
tree
for <2-3>*4



Summary

- Lexical and Syntactic Levels of Structure
  - Lexical – regular expressions and automata
  - Syntactic – grammars
- Grammar ambiguities
  - Hacked grammars
  - Abstract syntax trees
- Generation versus Recognition Approaches
  - Generation more convenient for specification
  - Recognition required in implementation

Handling If Then Else

Start → *Stat*

Stat → if *Expr* then *Stat* else *Stat*

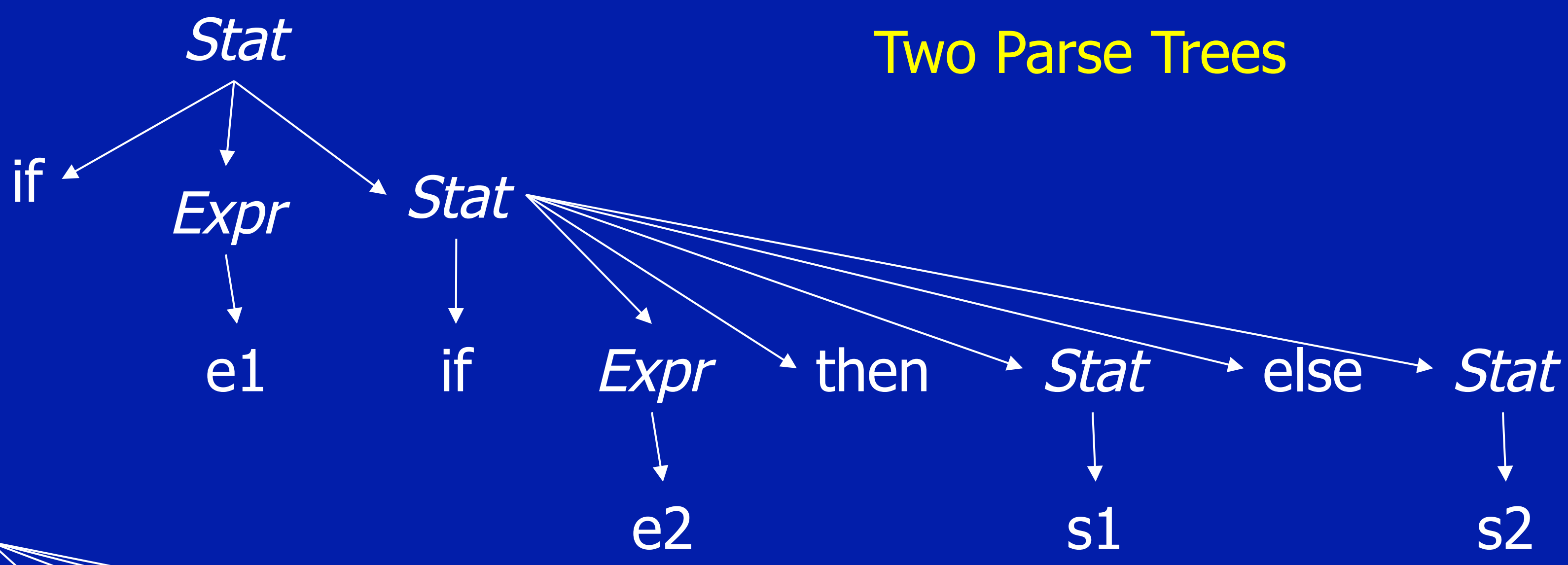
Stat → if *Expr* then *Stat*

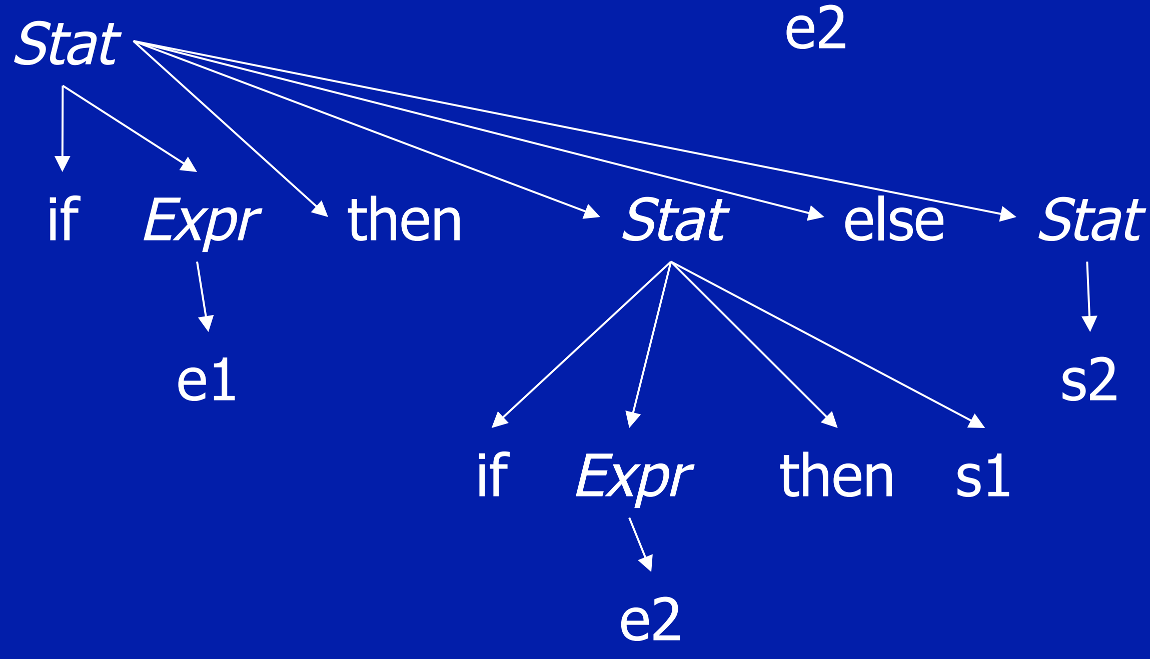
Stat → ...

Parse Trees

- Consider Statement if e_1 then if e_2 then s_1 else s_2

Two Parse Trees





Which is correct?

Alternative Readings

- Parse Tree Number 1

  if e_1

      if e_2 s_1

      else s_2

- Parse Tree Number 2

  if e_1

      if e_2 s_1

  else s_2

Grammar is ambiguous

Hacked Grammar

Goal → *Stat*

Stat → *WithElse*

Stat → *LastElse*

WithElse → if *Expr* then *WithElse* else *WithElse*

WithElse → <statements without if then or if then else>

LastElse → if *Expr* then *Stat*

LastElse → if *Expr* then *WithElse* else *LastElse*

Hacked Grammar

- Basic Idea: control carefully where an if without an else can occur
  - Either at top level of statement
  - Or as very last in a sequence of if then else if then ... statements

Grammar Vocabulary

- Leftmost derivation
  - Always expands leftmost remaining nonterminal
  - Similarly for rightmost derivation
- Sentential form
  - Partially or fully derived string from a step in valid derivation
  - 0 + *Expr* Op *Expr*
  - 0 + *Expr* - 2

Defining a Language

- Grammar
  - Generative approach
  - All strings that grammar generates (How many are there for grammar in previous example?)
- Automaton
  - Recognition approach
  - All strings that automaton accepts
- Different flavors of grammars and automata
- In general, grammars and automata correspond

Regular Languages

- Automaton Characterization
  - (S, A, F, s_0, s_F)
  - Finite set of states S
  - Finite Alphabet A
  - Transition function $F : S \times A \rightarrow S$
  - Start state s_0
  - Final states s_F
- Lanuage is set of strings accepted by Automaton

Regular Languages

- Regular Grammar Characterization
  - (T, NT, S, P)
  - Finite set of Terminals T
  - Finite set of Nonterminals NT
  - Start Nonterminal S (goal symbol, start symbol)
  - Finite set of Productions $P: NT \rightarrow T \cup NT \cup T\ NT$
- Language is set of strings generated by grammar

Grammar and Automata Correspondence

Grammar	Automaton
Regular Grammar	Finite-State Automaton
Context-Free Grammar	Push-Down Automaton
Context-Sensitive Grammar	Turing Machine

Context-Free Grammars

- Grammar Characterization
  - (*T,NT,S,P*)
  - Finite set of Terminals *T*
  - Finite set of Nonterminals *NT*
  - Start Nonterminal *S* (goal symbol, start symbol)
  - Finite set of Productions $P\colon NT \rightarrow (T \mid NT)^*$
- RHS of production can have any sequence of terminals or nonterminals

Push-Down Automata

- DFA Plus a Stack
  - (S, A, V, F, s_0, s_F)
  - Finite set of states S
  - Finite Input Alphabet A, Stack Alphabet V
  - Transition relation $F : S \times (A \cup \{\varepsilon\}) \times V \rightarrow S \times V^*$
  - Start state s_0
  - Final states s_F
- Each configuration consists of a state, a stack, and remaining input string

CFG Versus PDA

- CFGs and PDAs are of equivalent power
- Grammar Implementation Mechanism:
    - Translate CFG to PDA, then use PDA to parse input string
    - Foundation for bottom-up parser generators

Context-Sensitive Grammars and Turing Machines

- Context-Sensitive Grammars Allow Productions to Use Context
  - $P: (T.NT)+ \rightarrow (T.NT)^*$
- Turing Machines Have
  - Finite State Control
  - Two-Way Tape Instead of A Stack