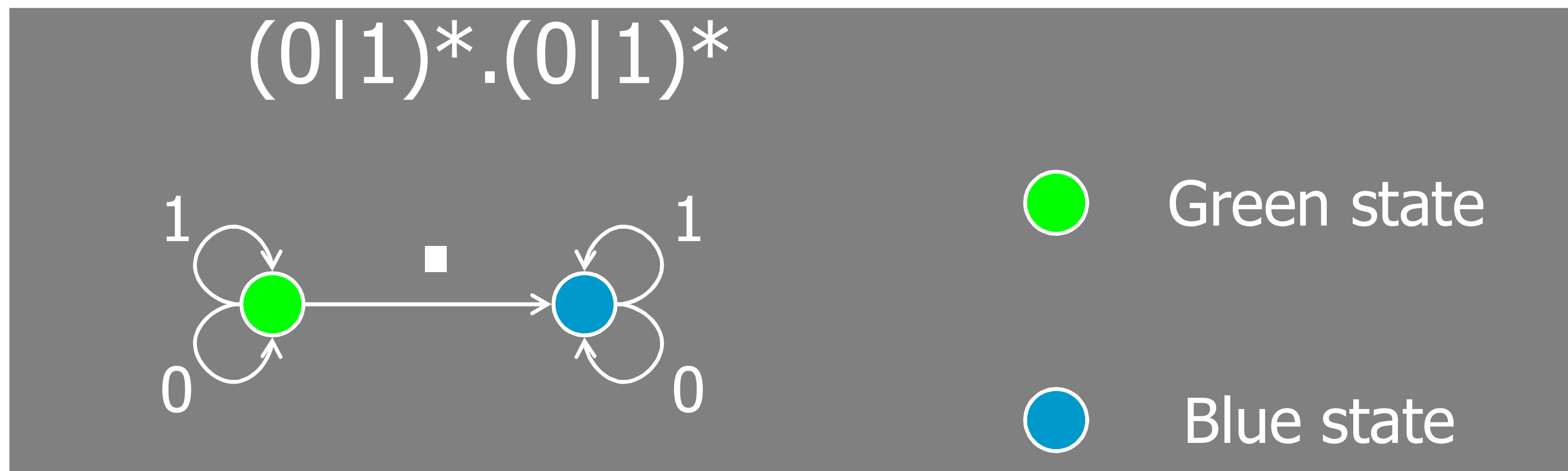
Finite-State Automata

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

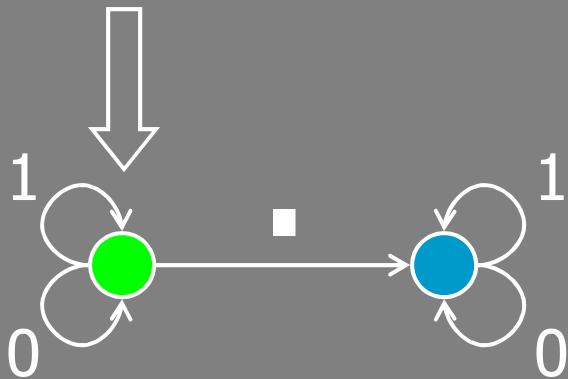


Automaton

- String recognition
  - Start with start state and first letter of 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
- The language of automaton is the set of strings it accepts

Example

Current state



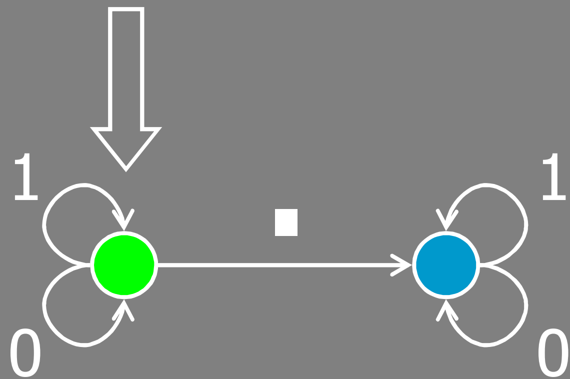
Start state

Accept state

1 1 . 0

Example

Current state



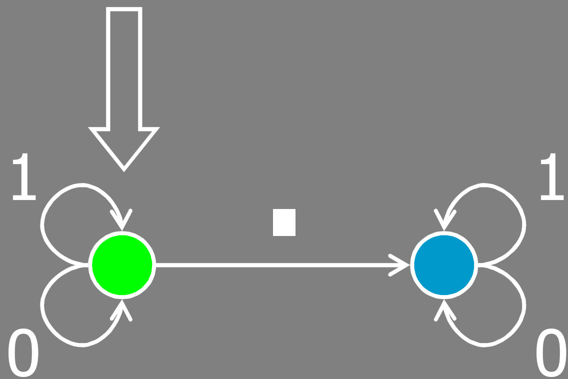
Start state

Accept state

1 1 . 0

Example

Current state



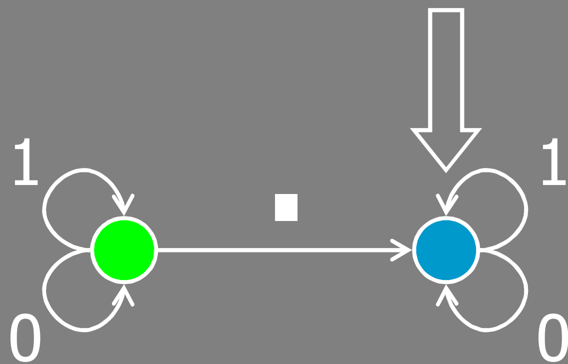
Start state

Accept state

1 1 . 0

Example

Current state



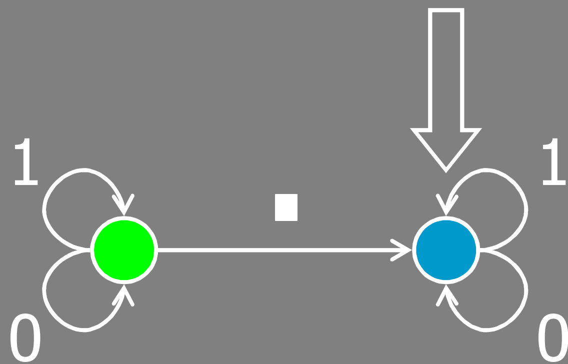
Start state

Accept state

1 1 . 0

Example

Current state



Start state

Accept state

1 1 . 0

String is accepted!

Generation Versus Recognition

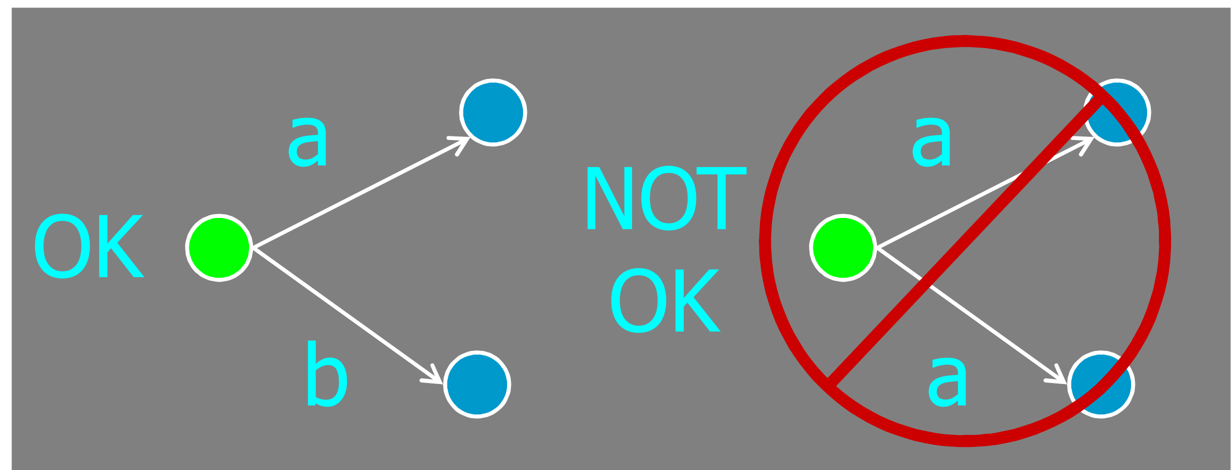
- Syntactically a language is a set of strings
- Regular expressions
  - Defining a language by composition
- Automata
  - Defining a language by implementation
    - Determining whether a string is in the language or not
  - Theoretically equivalent (for regular expressions and automata)

Can we compile the Regular Expression Language into an Automata Implementation?

Yes we can!

NFA vs. DFA

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



- NFA – can do both
  - It can be in many states at the same time
  - accept if ANY of its current states is an accepting state

RE to Automata

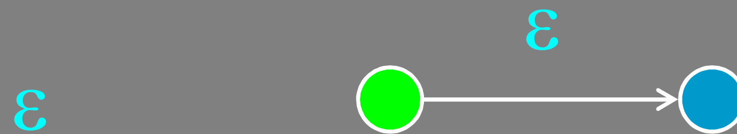
- Construction by structural induction
- Convert regular expression into an automaton with
  - One start state
  - One accept state
- Assume any sub-expressions have been converted
  - this works as long as you do it bottom up

RE to Automata

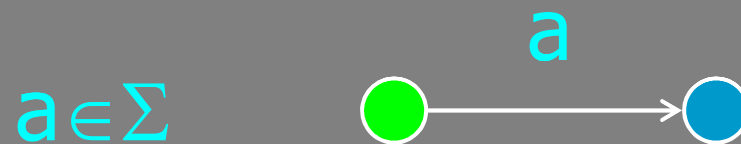
Start state

Accept state

ε



$a \in \Sigma$

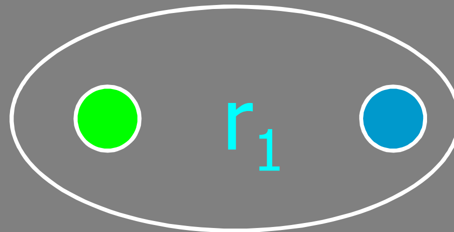


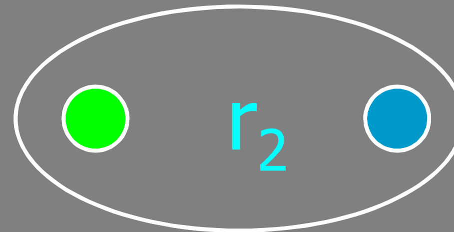
Sequence

Start state

Accept state

r_1r_2





Sequence

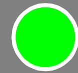
Old start state

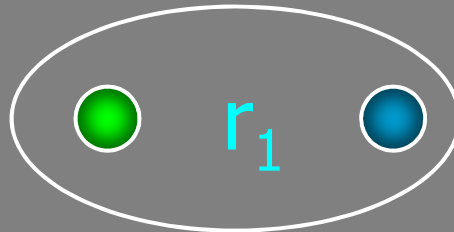
Start state

Old accept state

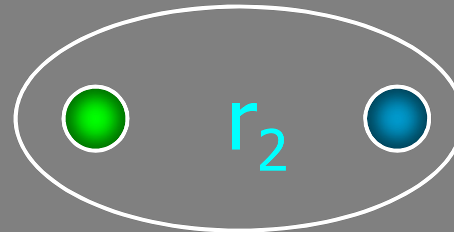
Accept state

$r_1 r_2$





r_1

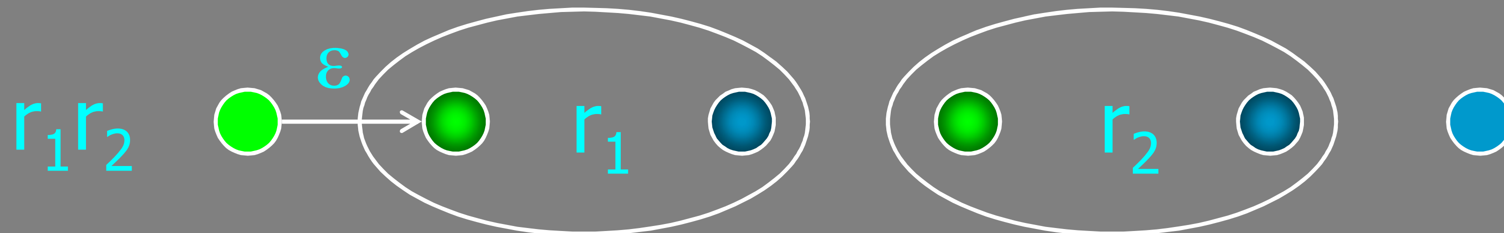


r_2

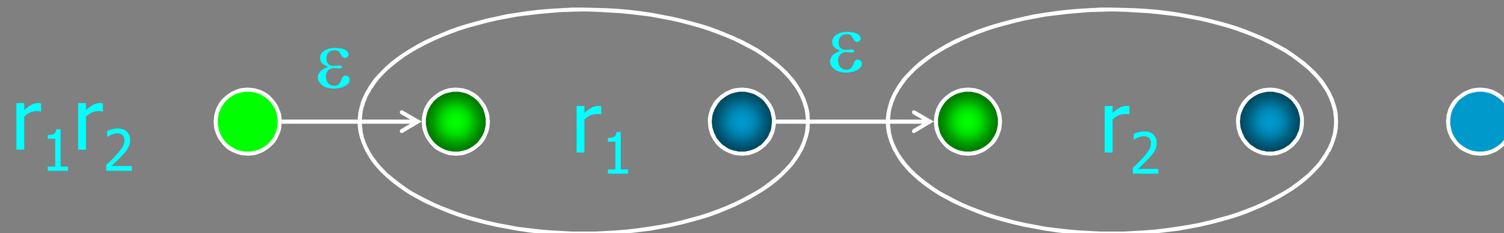
Sequence

Old start state | Start state

Old accept state | Accept state

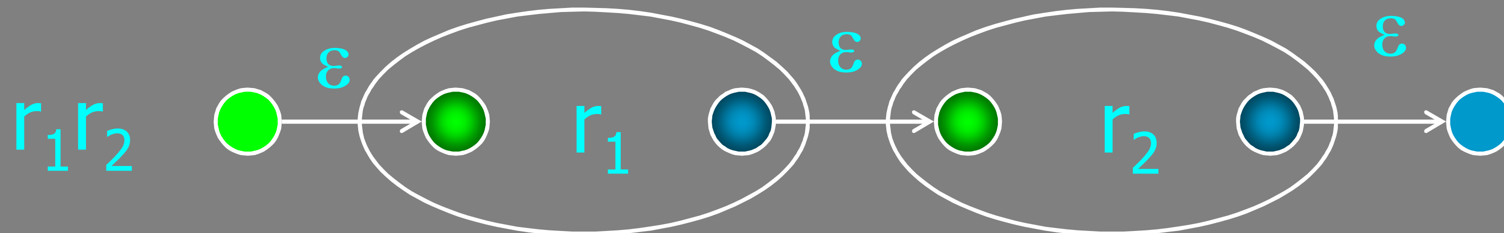


Sequence



Sequence

Old start state
Start state
Old accept state
Accept state

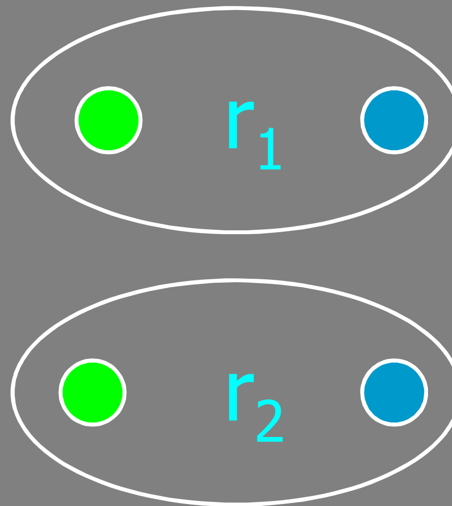


Choice

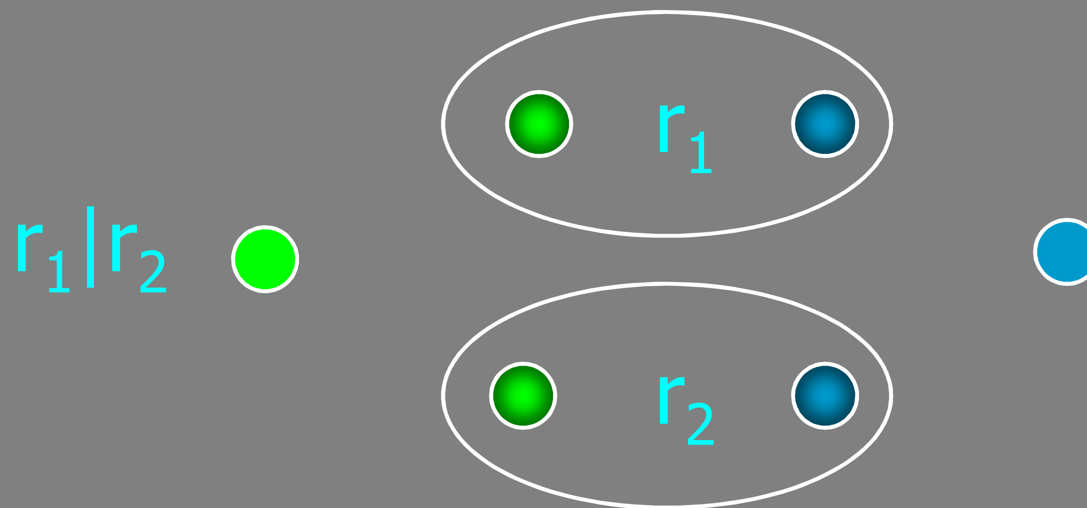
Start state

Accept state

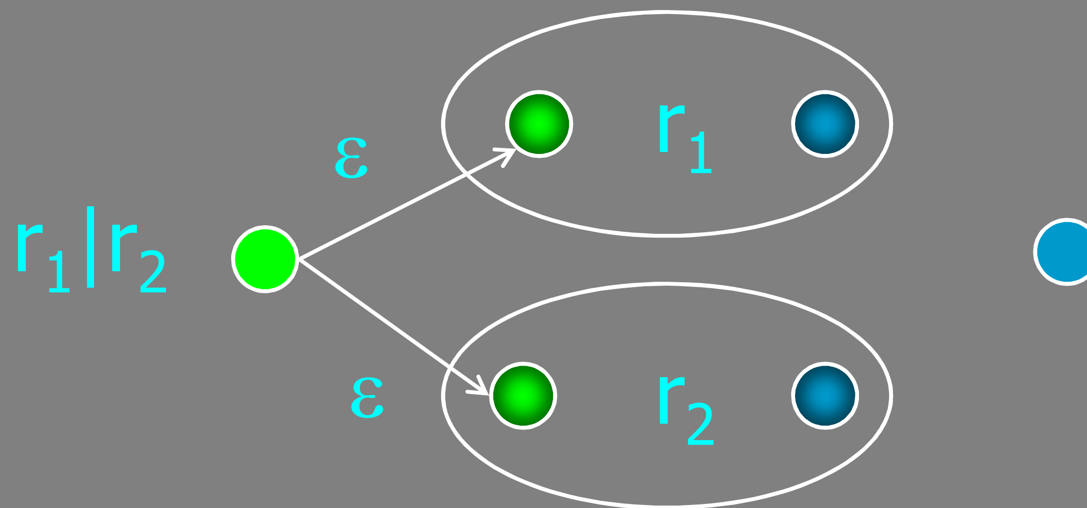
$r_1|r_2$



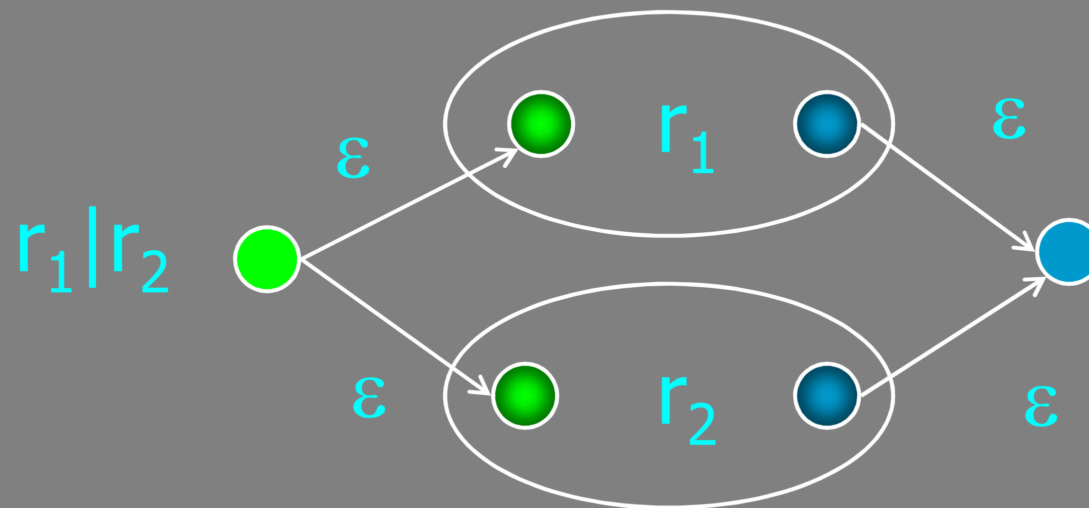
Choice



Choice



Choice



Kleene Star

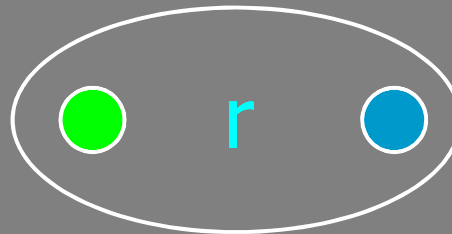
Old start state

Start state

Old accept state

Accept state

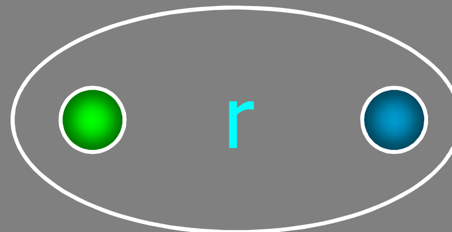
r*



Kleene Star

Old start state
Start state
Old accept state
Accept state

r*



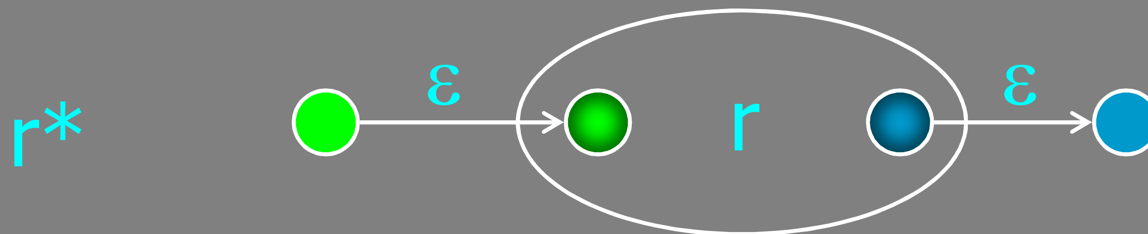
Kleene Star

Old start state

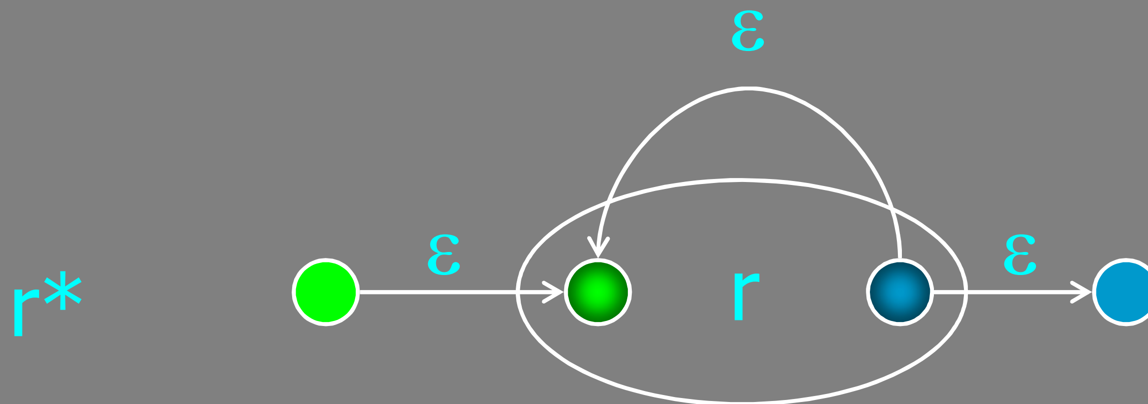
Start state

Old accept state

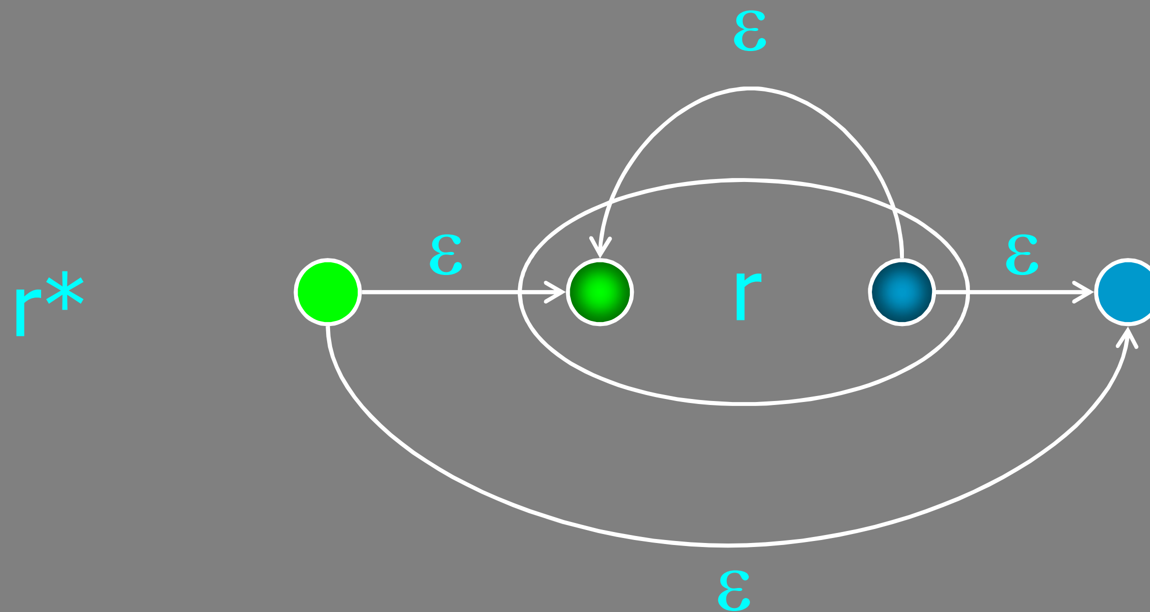
Accept state



Kleene Star



Kleene Star



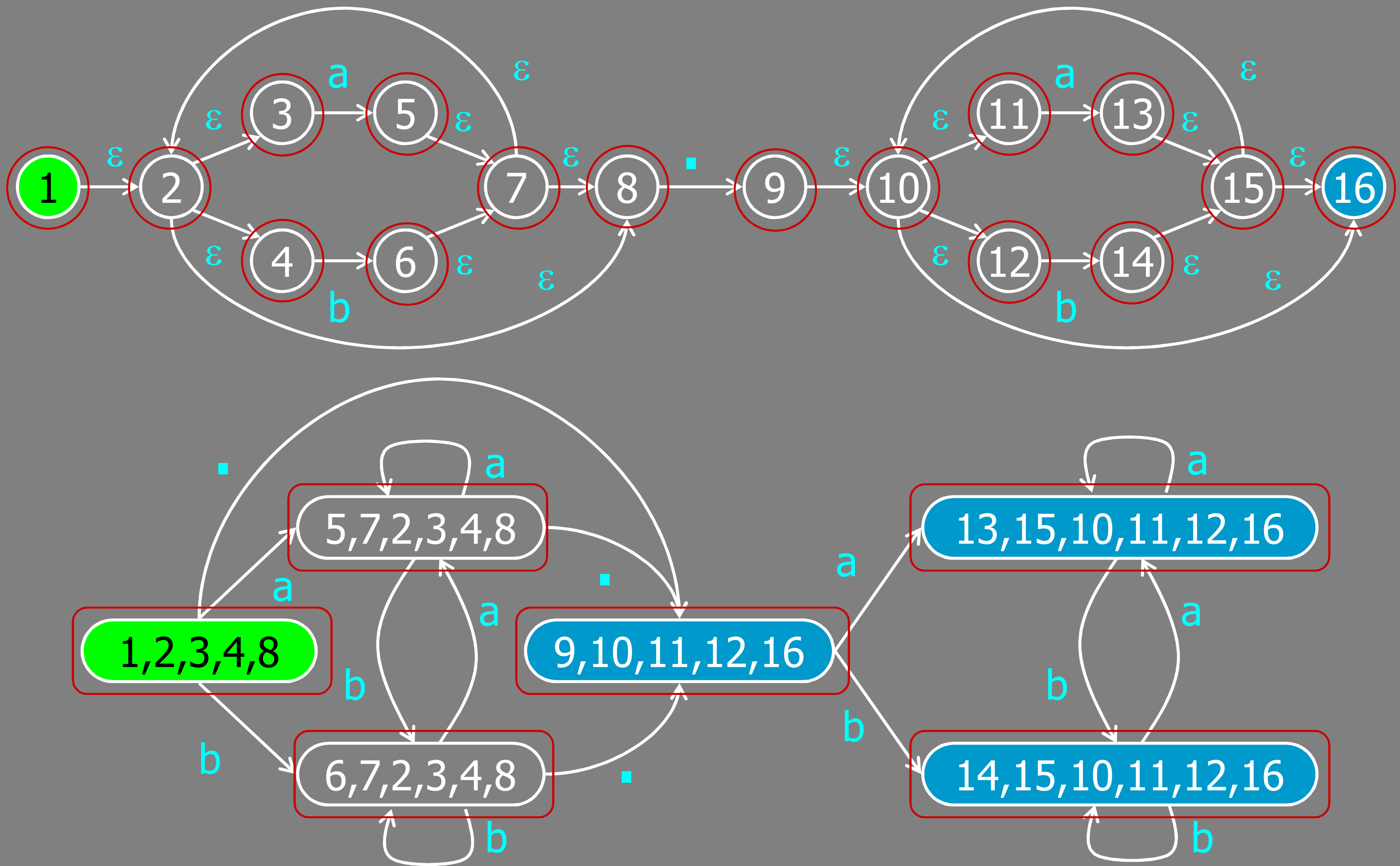
Conversions

- 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)
  - Simple algorithm available (check a text book for details)

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

- Typical classes of tokens:
  - 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
  - Define each class of tokens by a regular expression
- Also need lexical category for
  - comments
  - whitespace

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

Write a Regular Expression

- All strings of the wedge alphabet { <, > }
  - (<|>)*
- Strings with open wedges followed by close wedges
  - <*>*
- Strings with matching wedges
  - Not with a regular expression!

Nested Experessions

- Are regular languages sufficient for specifying syntax?
- Try the following
  - (a+(b-c))*(d-(x-(y-z)))
  - if (x < y) if (y < z) a = 5 else a = 6 else a = 7

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

Generation

start with *Start* nonterminal

repeat

choose a nonterminal

choose a production with that nonterminal in LHS

replace nonterminal with RHS of production

until no more nonterminals in the string

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

Sample Derivation

Op = +|-|*|/

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

Open = <

Close = >

1) *Start* $\rightarrow$ *Expr*

2) *Expr* $\rightarrow$ *Expr* Op *Expr*

3) *Expr* $\rightarrow$ Int

4) *Expr* $\rightarrow$ Open *Expr* Close

Open Int Op Int Close Op Int

< Int Op Int Close Op Int

< [0-9][0-9]* Op Int Close Op Int

< 2 Op Int Close Op Int

< 2 +|-|*|/ Int Close Op Int

< 2 - Int Close Op Int

< 2 - [0-9][0-9]* Close Op Int

< 2 - 1 Close Op Int

< 2 - 1 > Op Int

< 2 - 1 > +|-|*|/ Int

< 2 - 1 > + Int

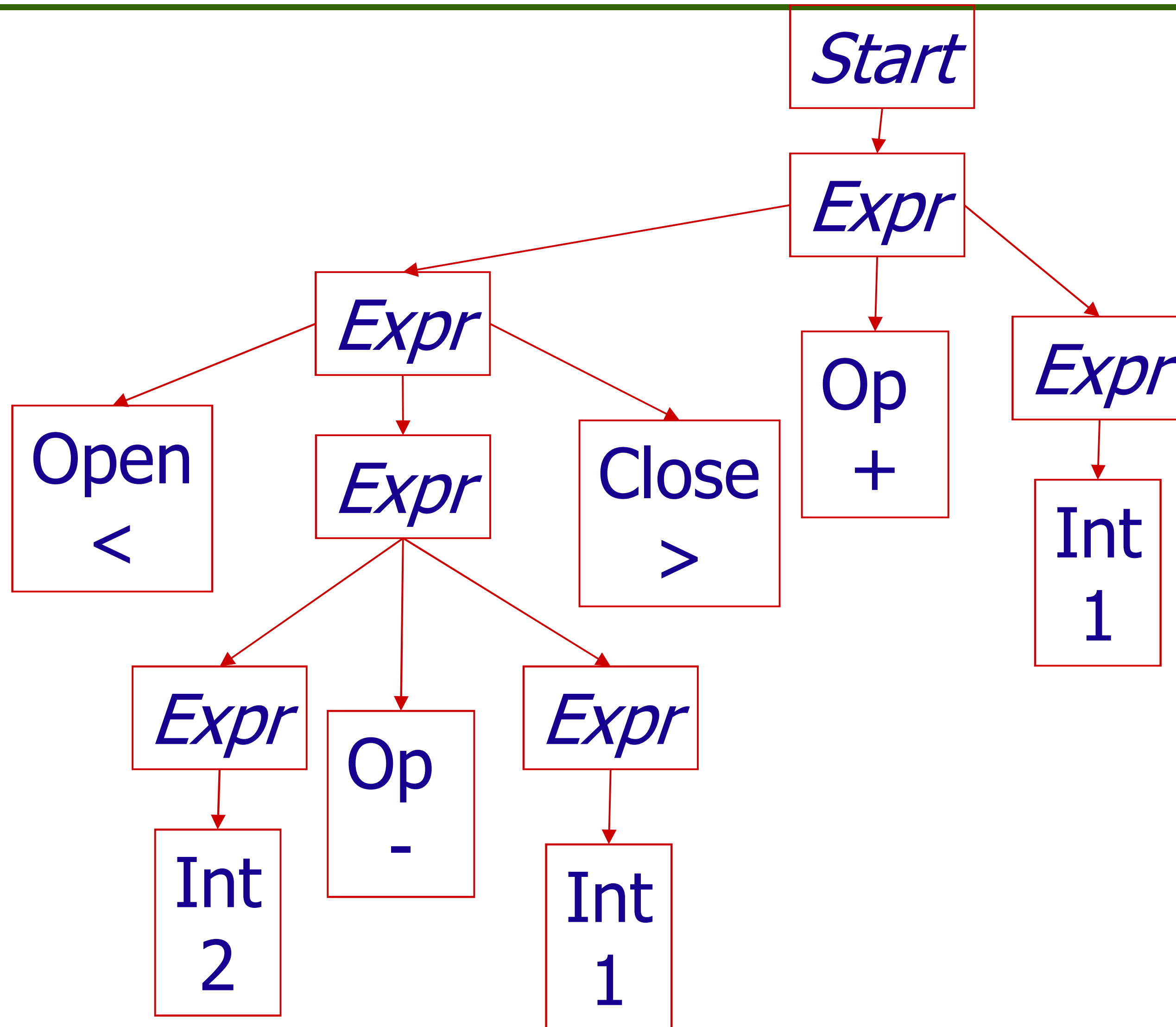
< 2 - 1 > + [0-9][0-9]*

< 2 - 1 > + 1

Parse Tree

- Tracks the history of a derivation
- Internal Nodes: Nonterminals
- Leaves: Terminals
- Edges: individual derivations

Parse Tree for <2-1>+1



Parsing

Op = +|-|*|/

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

Open = <

Close = >

1) *Start* → *Expr*

2) *Expr* → *Expr* Op *Expr*

3) *Expr* → Int

4) *Expr* → Open *Expr* Close

< 2 - 1 > + 1

Open 2 - 1 > + 1

Open Int - 1 > + 1

Open Int Op 1 > + 1

Open Int Op Int > + 1

Open Int Op Int Close + 1

Open Int Op Int Close Op 1

Open Int Op Int Close Op Int

Parsing

Op = +|-|*|/

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

Open = <

Close = >

1) *Start* → *Expr*

2) *Expr* → *Expr* Op *Expr*

3) *Expr* → Int

4) *Expr* → Open *Expr* Close

Open Int Op Int Close Op Int

Open *Expr* Op Int Close Op Int

Open *Expr* Op *Expr* Close Op Int

Open *Expr* Close Op Int

Expr Op Int

Expr Op *Expr*

Expr

Start

What is this Grammar?

What is the language of the following grammar?

$Start \rightarrow S$

$S \rightarrow (L)$

$S \rightarrow \mathrm{a}$

$L \rightarrow L , S$

$L \rightarrow S$

$Start \to S$

$S \to (L)$

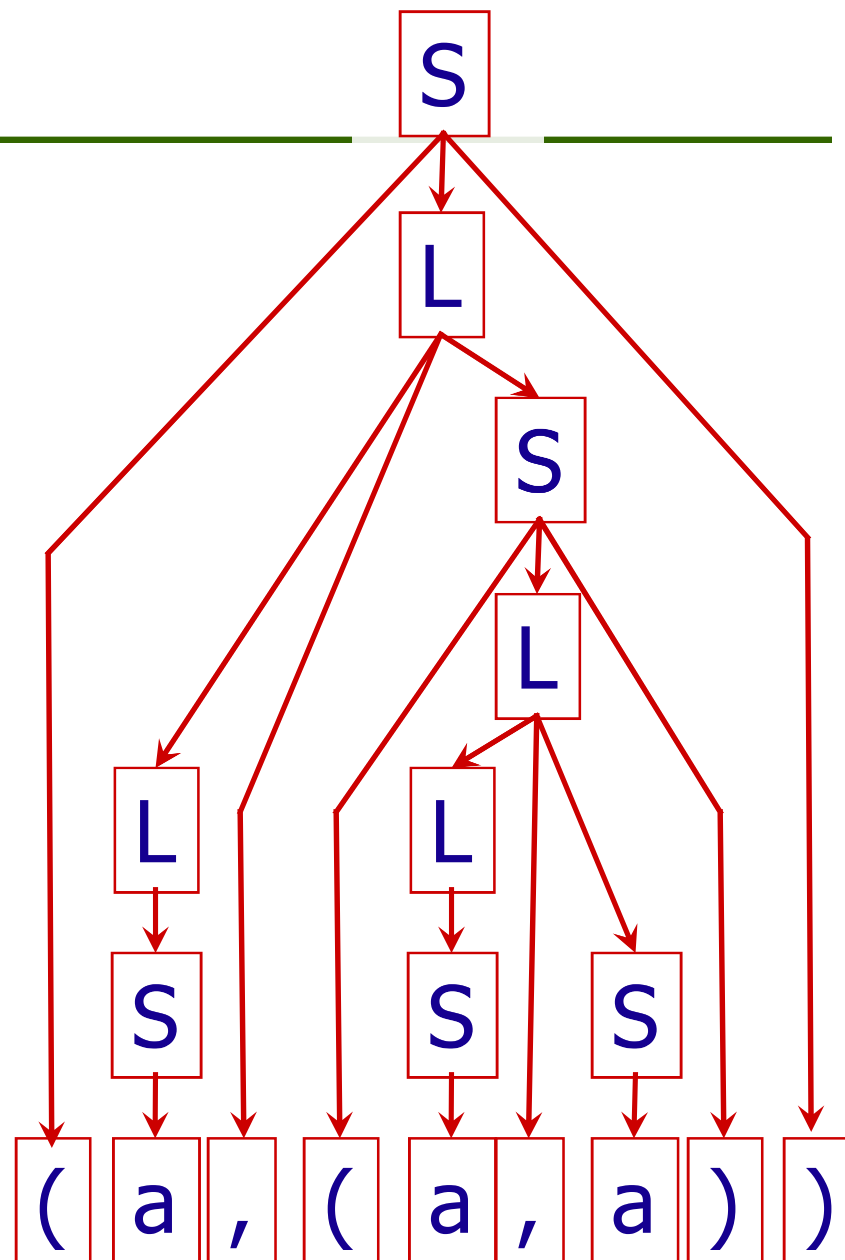
$S \to$ a

$L \to L , S$

$L \to S$

Write a parse tree for

(a, (a, a))

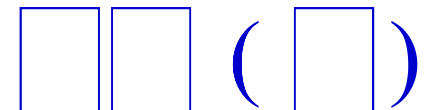


Terminology

- Many different parsing techniques
  - Each can handle some set of CFGs
  - Categorization of techniques

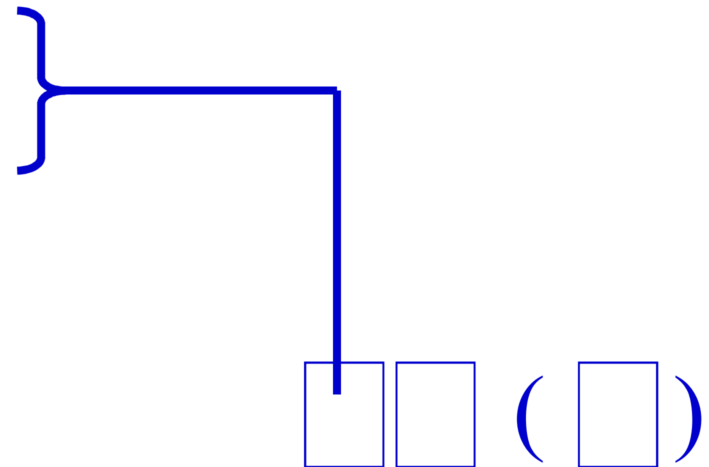
Terminology

- Many different parsing techniques
  - Each can handle some set of CFGs
  - Categorization of techniques



Terminology

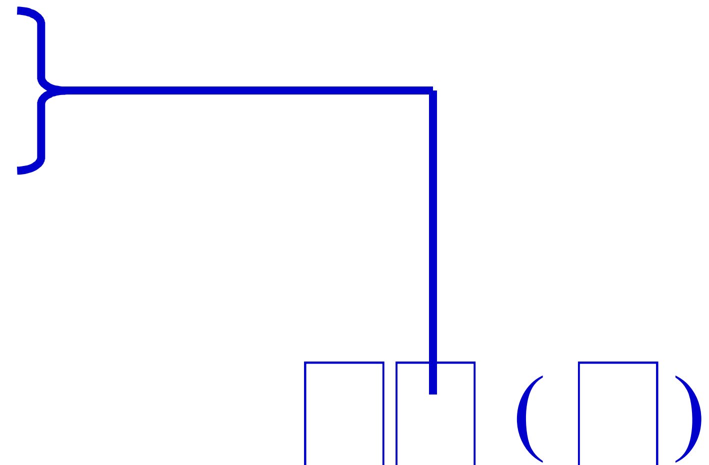
- Many different parsing techniques
  - Each can handle some set of CFGs
  - Categorization of techniques
  - **L** - parse from left to right
  - **R** - parse from right to left



Terminology

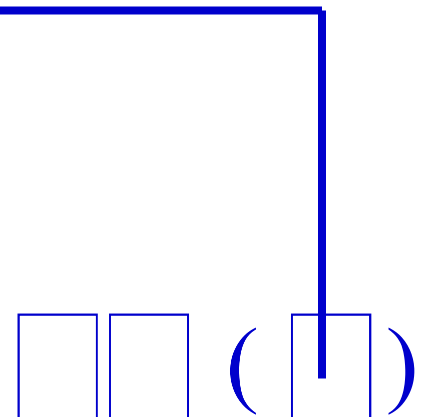
- Many different parsing techniques
  - Each can handle some set of CFGs
  - Categorization of techniques

  - **L** - leftmost derivation
  - **R** - rightmost derivation



Terminology

- Many different parsing techniques
  - Each can handle some set of CFGs
  - Categorization of techniques
  - Number of lookahead characters



Terminology

- Many different parsing techniques
  - Each can handle some set of CFGs
  - Categorization of techniques
  - Examples: LL(0), LR(1)
- Today: Building a LL(k) parser
  - Manual construction of a recursive descent parser
    - Code parser as set of mutually recursive procedures
    - Structure of parser matches structure of grammar

L R (k)

Ambiguity

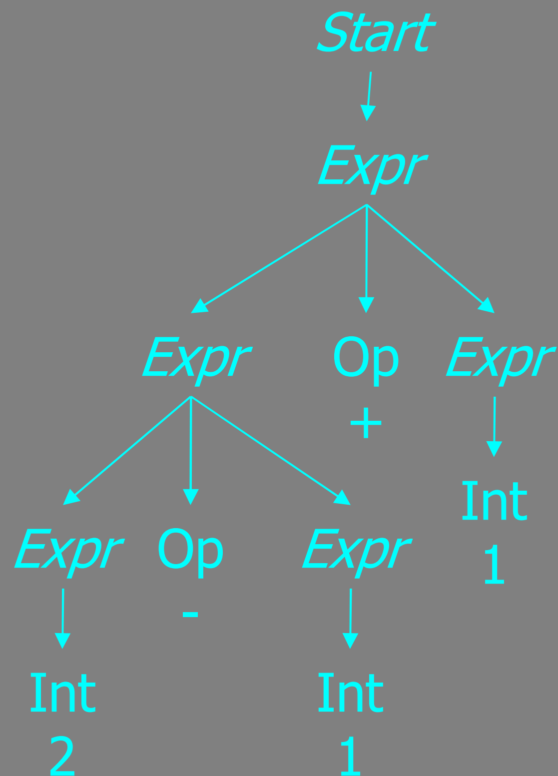
Ambiguity in Grammar

- 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)

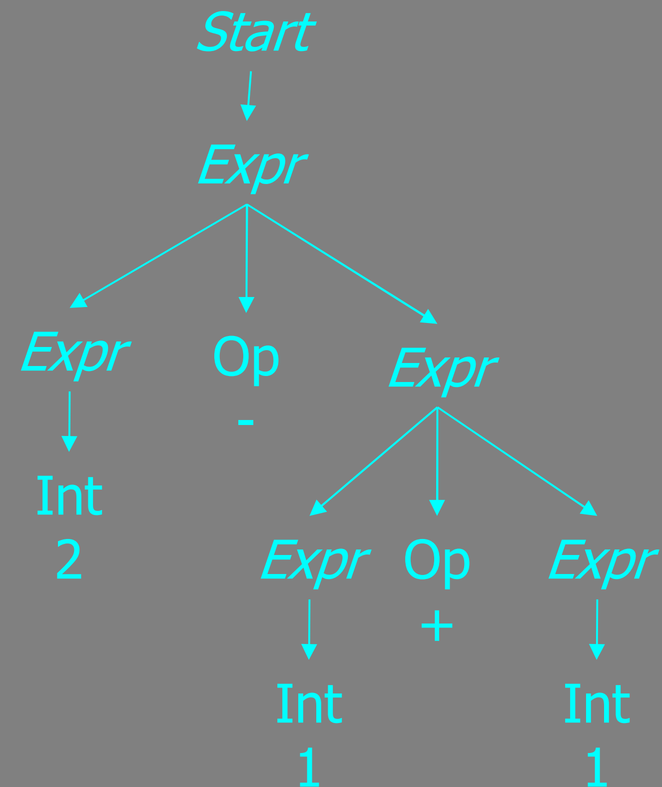
Ambiguity Example

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

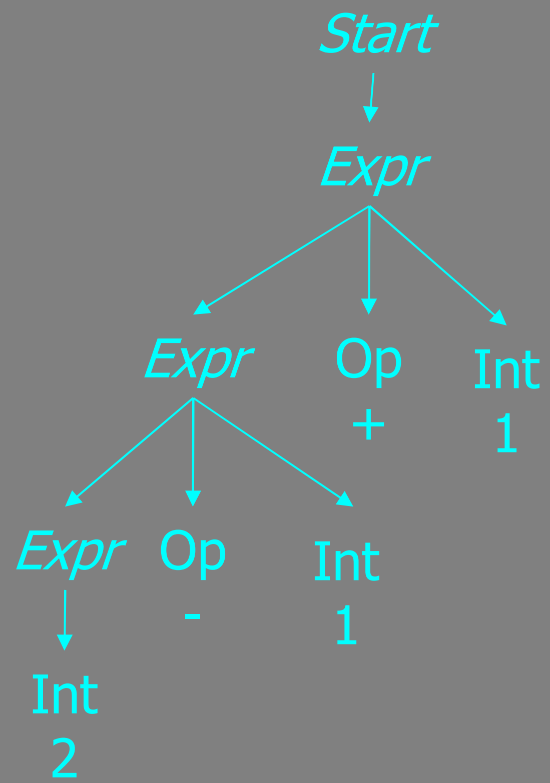
Original Grammar	Hacked Grammar
Start → *Expr*	*Start* → *Expr*
Expr → *Expr* Op *Expr*	*Expr* → *Expr* Op Int
Expr → Int	*Expr* → Int
Expr → Open *Expr* Close	*Expr* → Open *Expr* Close

Conceptually, makes all operators associate to left

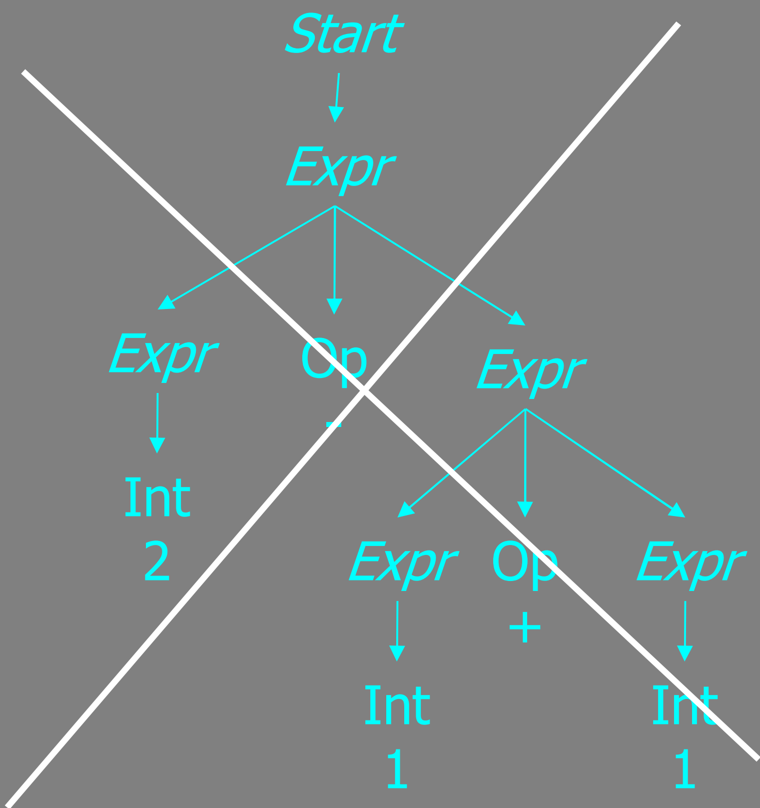
Parse Trees for Hacked Grammar

Only one parse tree for 2-1+1!

Valid parse tree



No longer valid parse tree



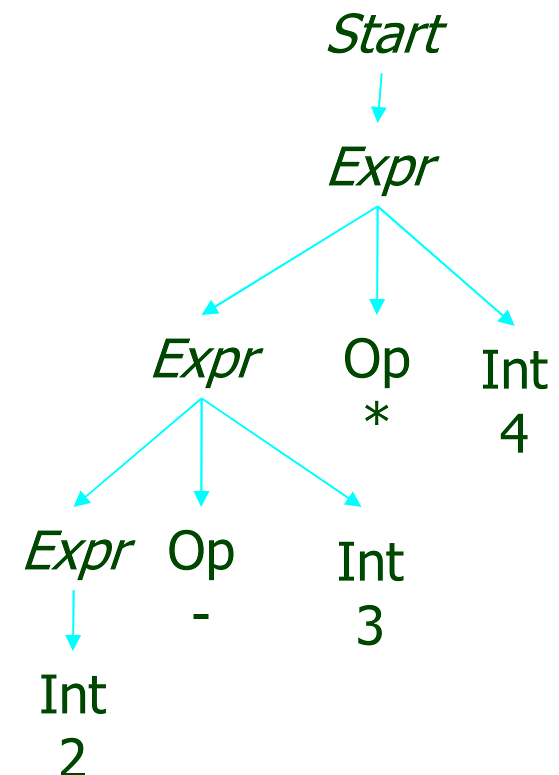
Exercise

- Construct a grammar for this language:

Expressions involving Ints, +, -, *, /

Precedence Violations

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



Hacking Around Precedence

Original Grammar

Op = +|-|*|/

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

Open = <

Close = >

Start $\rightarrow$ *Expr*

Expr $\rightarrow$ *Expr* Op Int

Expr $\rightarrow$ Int

Expr $\rightarrow$ Open *Expr* Close

Hacked Grammar

AddOp = +|-

MulOp = *|/

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

Open = <

Close = >

Start $\rightarrow$ *Expr*

Expr $\rightarrow$ *Expr* AddOp *Term*

Expr $\rightarrow$ *Term*

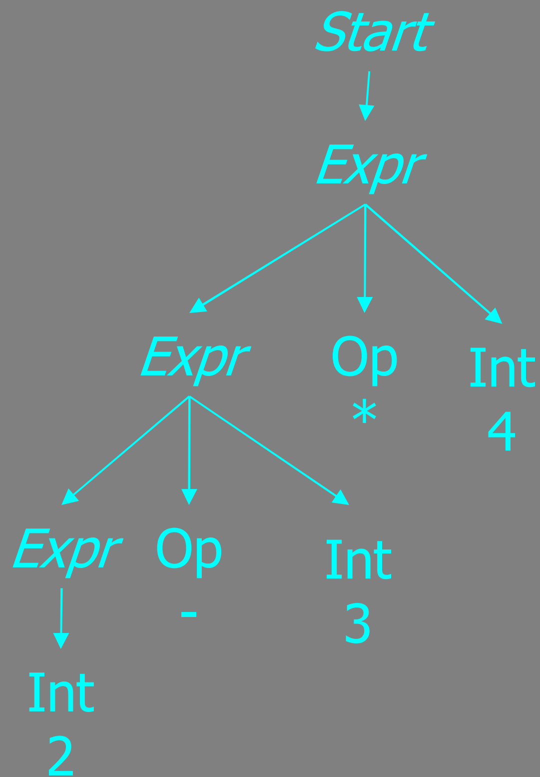
Term $\rightarrow$ *Term* MulOp *Num*

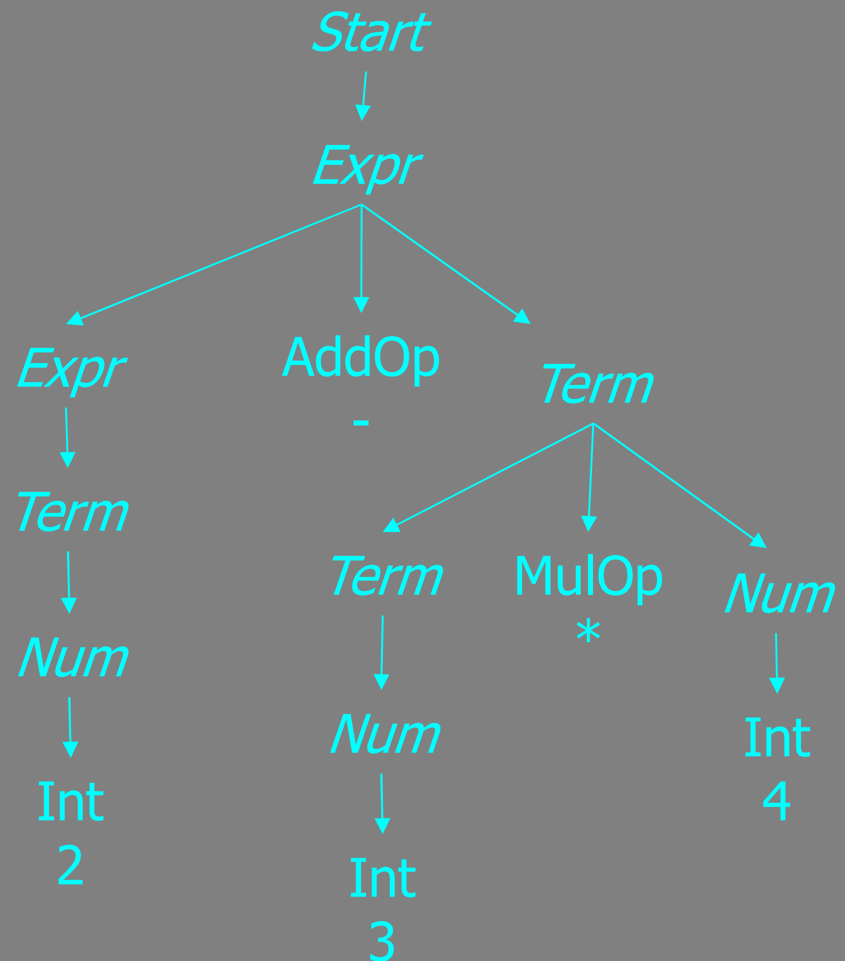
Term $\rightarrow$ *Num*

Num $\rightarrow$ Int

Num $\rightarrow$ Open *Expr* Close

Parse Tree Changes





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

What is different?

Hacked Grammar I	Hacked Grammar II
AddOp = +|-	AddOp = +|-
MulOp = *|/	MulOp = *|/
Int = [0-9] [0-9]*	Int = [0-9] [0-9]*
Open = <	Open = <
Close = >	Close = >
Start → *Expr*	*Start* → *Expr*
Expr → *Expr* AddOp *Term*	*Expr* → *Expr* AddOp *Term*
Expr → *Term*	*Expr* → *Term*
Term → *Term* MulOp *Num*	***Expr* → Open *Expr* Close**
Term → *Num*	*Term* → *Term* MulOp *Num*
Num → Int	*Term* → *Num*
***Num* → Open *Expr* Close**	*Num* → Int

Handling If Then Else

Start → *Stat*

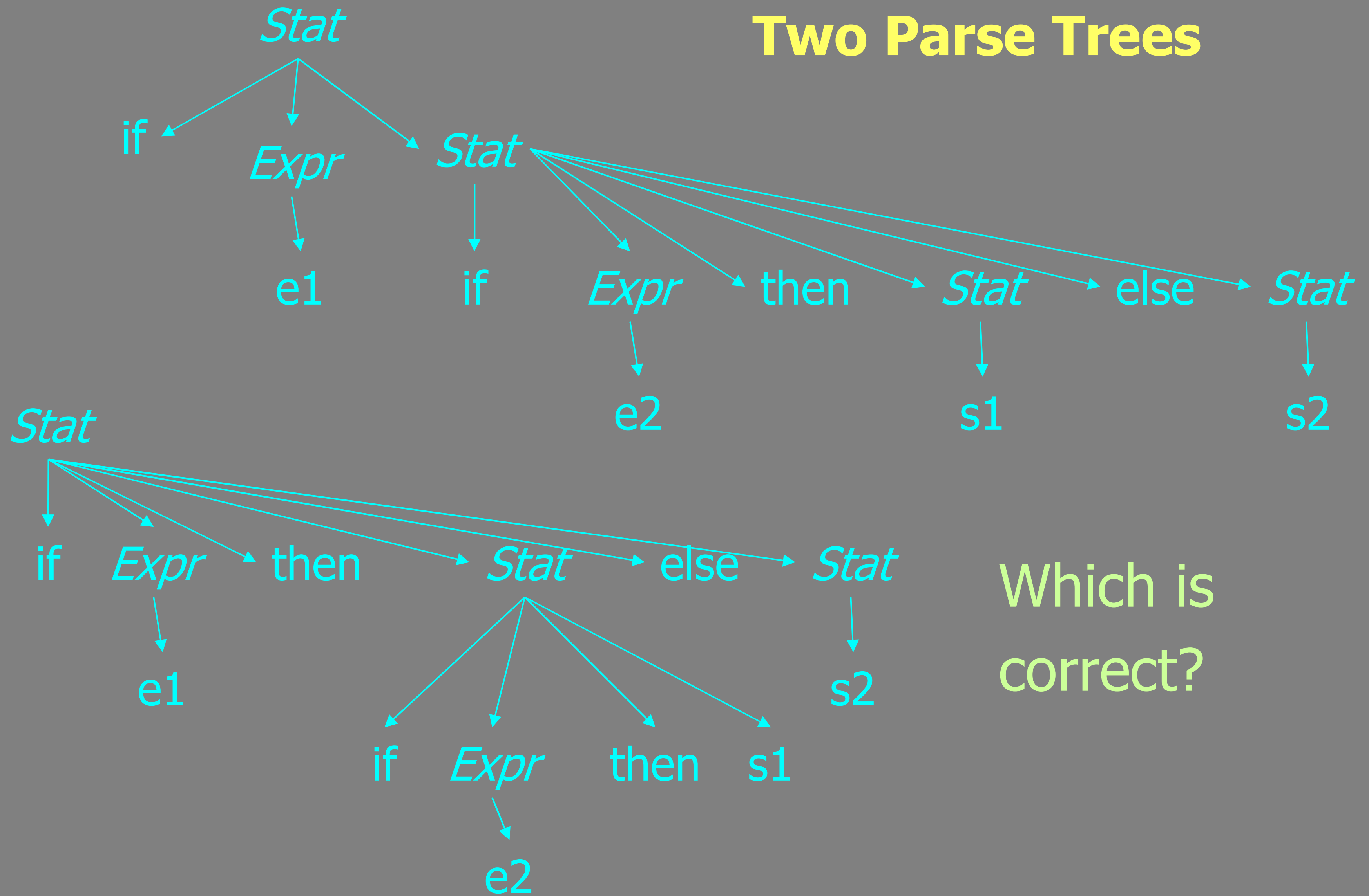
Stat → if *Expr* then *Stat* else *Stat*

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
Stat
if
Expr
Stat
e1
if
Expr
then
Stat
else
Stat
e2
s1
s2
Stat
if
Expr
then
Stat
else
Stat
e1
s2
if
Expr
then
s1
e2
Which is correct?

Alternative Readings

- Parse Tree Number 1

  if e_1

      if e_2

          s_1

      else

          s_2

  ← Programming languages prefer this

- Parse Tree Number 2

  if e_1

      if e_2

          s_1

  else

      s_2

Grammar is ambiguous

This is know as the dangling-else problem

Resolving the dangling-else

- Rule: Associate else with the closest if
  - Suppose you have If e then S1 else S2
    - Can S1 = If e then S3 ?
    - NO! That would violate our rule.
    - An ifStmt inside S1 must have an else part
      - (Or be inside a block)
- This can be encoded in the grammar.

Hacked Grammar

Start → *Stmt*

Stmt → if *Expr* then *Stmt*

Stat → if *Expr* then *ThenStmt* else *Stmt*

Stmt → OtherStmt

ThenStmt → if *Expr* then *ThenStmt* else *ThenStmt*

ThenStmt → OtherStmt

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

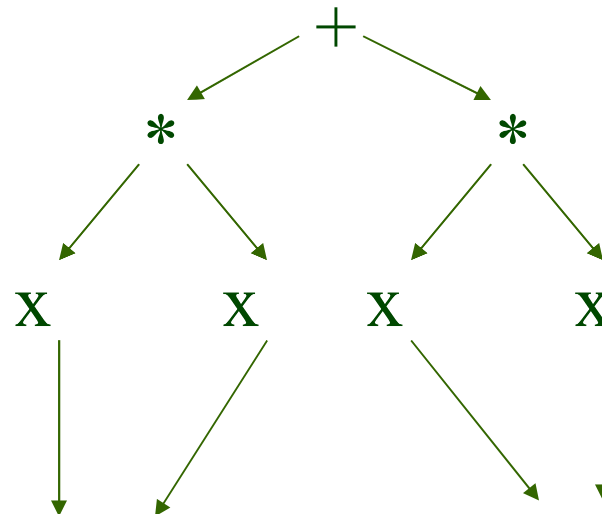
Abstract Syntax Trees

Abstract Versus Concrete Trees

- Remember grammar hacks
  - left factoring, ambuguity elimination, precedence of binary operators
- Hacks lead to a tree that may not reflect cleanest interpretation of program
- May be more convenient to work with abstract syntax tree (roughly, parse tree from grammar before hacks)

Example

x*x + y*y



Each node in the tree is a datastructure
Immutability can allow sharing

Example

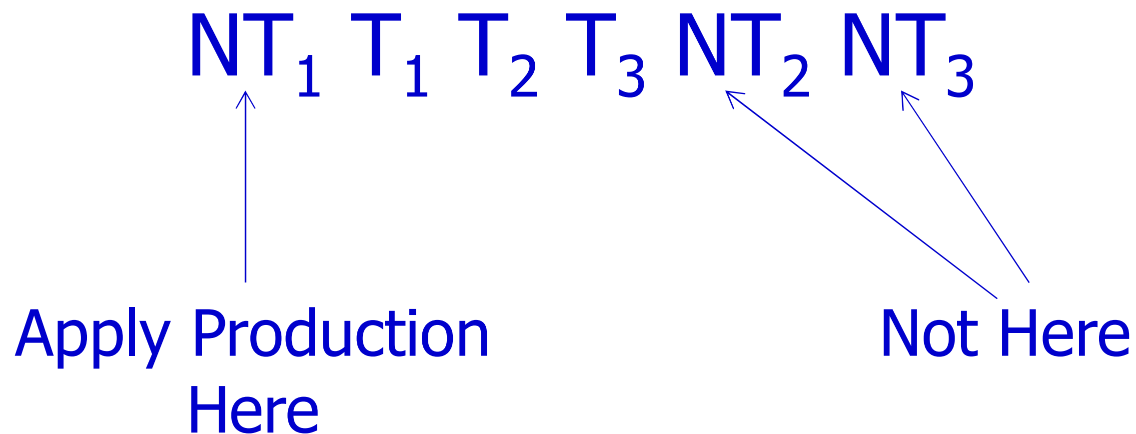
```
class eBinary : public Expression {
protected:
        const Expression* left_;
        const Expression* right_;
        const BinaryOperator op_;
public:
        eBinary(Expression* left, Expression* right, BinaryOperator op):
        Expression(Expression::BINOP),left_(left), right_(right), op_(op){}
        const Expression* left() {  return left_;  }
        const Expression* right() {  return right_; }
        const BinaryOperator op() {  return op_; }
        void accept(Visitor& v){
                v.visit(*this);
        }

};
```

Top Down Parsing

Basic Approach

- Start with Start symbol
- Build a leftmost derivation
  - If leftmost symbol is nonterminal, choose a production and apply it
  - If leftmost symbol is terminal, match against input
  - If all terminals match, have found a parse!
  - Key: find correct productions for nonterminals

Graphical Illustration of Leftmost Derivation



Grammar for Parsing Example

Start → *Expr*

Expr → *Expr* + *Term*

Expr → *Expr* - *Term*

Expr → *Term*

Term → *Term* * Int

Term → *Term* / Int

Term → Int

- Set of tokens is
  { +, -, *, /, Int },
  where Int = [0-9][0-9]*
  - For convenience, may represent each Int n token by n

Parsing Example

Parse Tree

Start

Remaining Input

2-2*2

Sentential Form

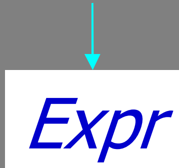
Start

Current Position in Parse Tree

Parsing Example

Parse Tree

Start



Current Position in Parse Tree

Remaining Input

2-2*2

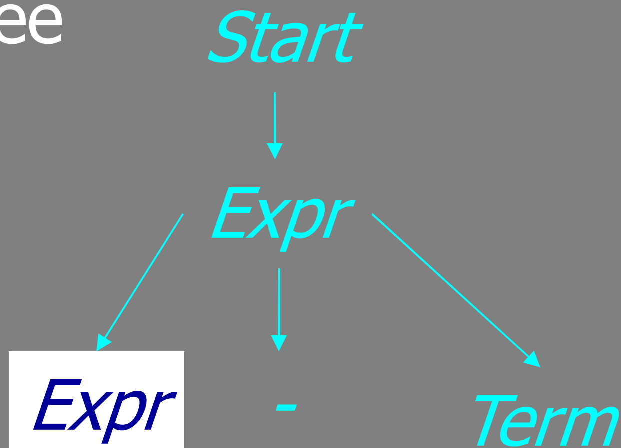
Sentential Form

Expr

Applied Production

Start → *Expr*

Parsing Example



Remaining Input

2-2*2

Sentential Form

Expr - Term

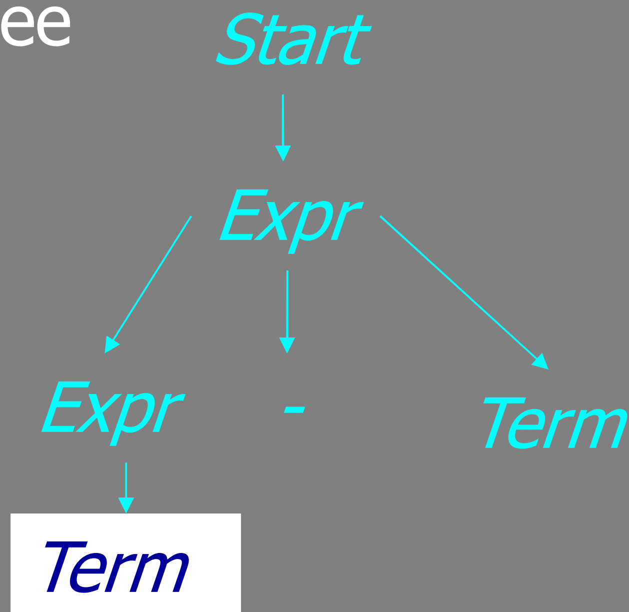
Expr $\rightarrow$ *Expr + Term*
Expr $\rightarrow$ *Expr - Term*
Expr $\rightarrow$ *Term*

Applied Production

Expr $\rightarrow$ *Expr - Term*

Parsing Example

Parse Tree



Expr → *Expr* + *Term*
Expr → *Expr* - *Term*
Expr → *Term*

Remaining Input

2-2*2

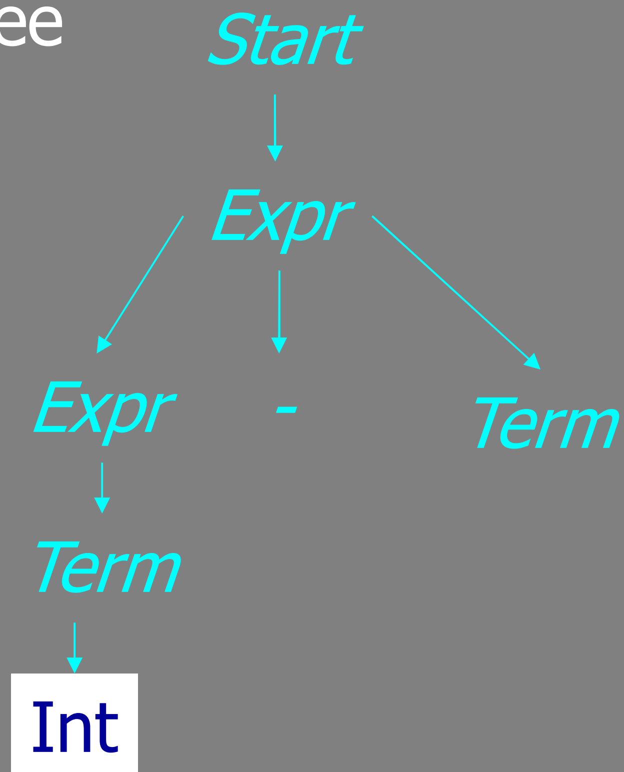
Sentential Form

Term - *Term*

Applied Production

Expr → *Term*

Parsing Example



Remaining Input

2-2*2

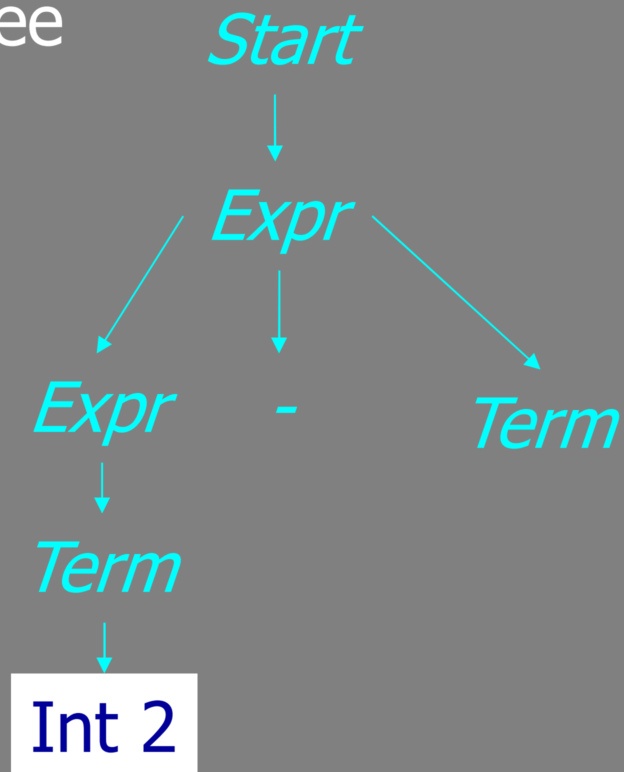
Sentential Form

Int - *Term*

Applied Production

Term → Int

Parsing Example



Match
Input
Token!

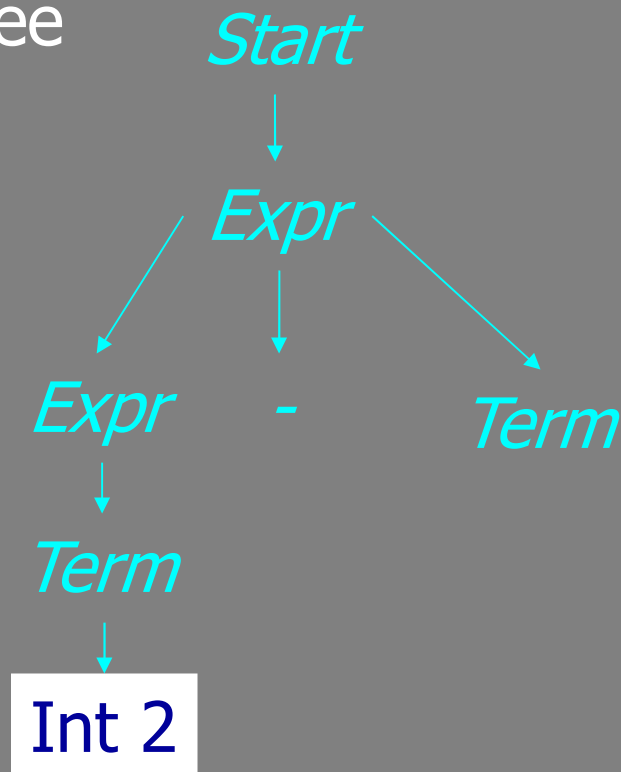
Remaining Input

2-2*2

Sentential Form

2 - *Term*

Parsing Example



Match
Input
Token!

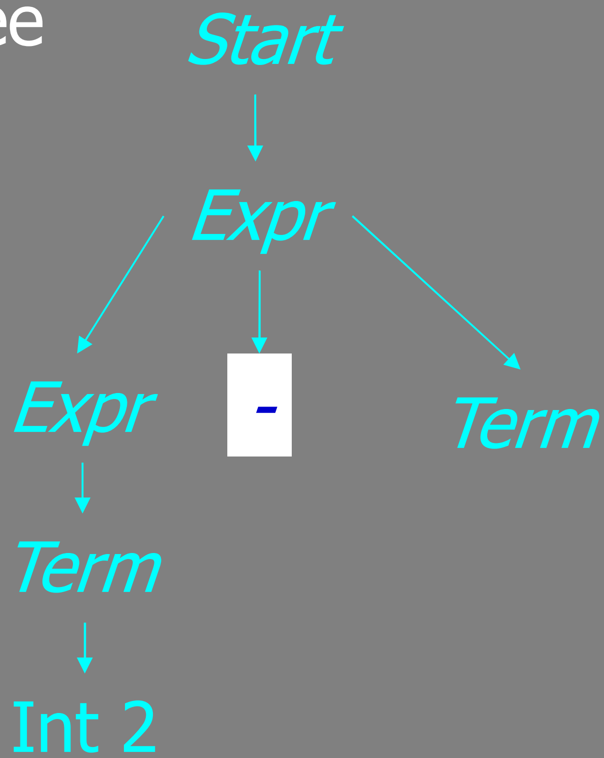
Remaining Input

-2*2

Sentential Form

2 - *Term*

Parsing Example



Match
Input
Token!

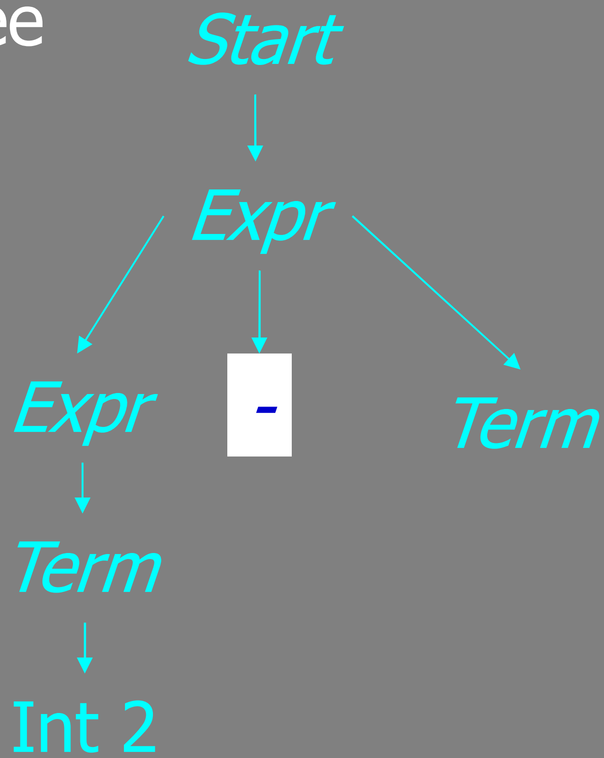
Remaining Input

-2*2

Sentential Form

2 - *Term*

Parsing Example



Match Input Token!

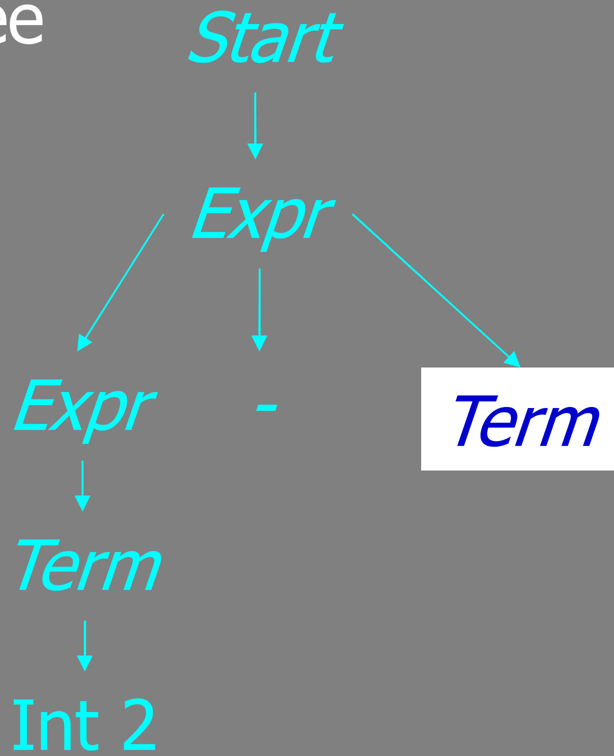
Remaining Input

2*2

Sentential Form

2 - *Term*

Parsing Example



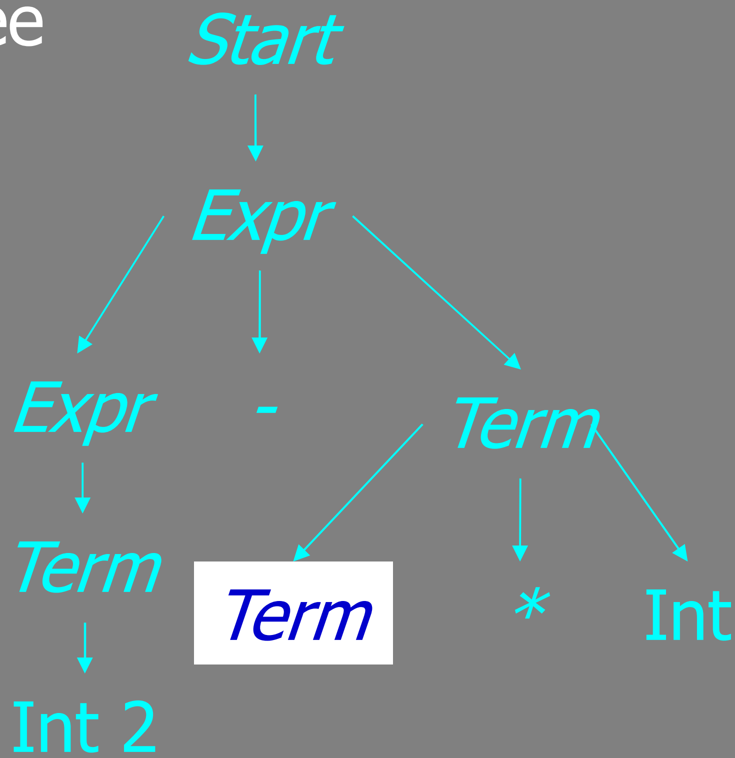
Remaining Input

2*2

Sentential Form

2 - *Term*

Parsing Example



Remaining Input

2*2

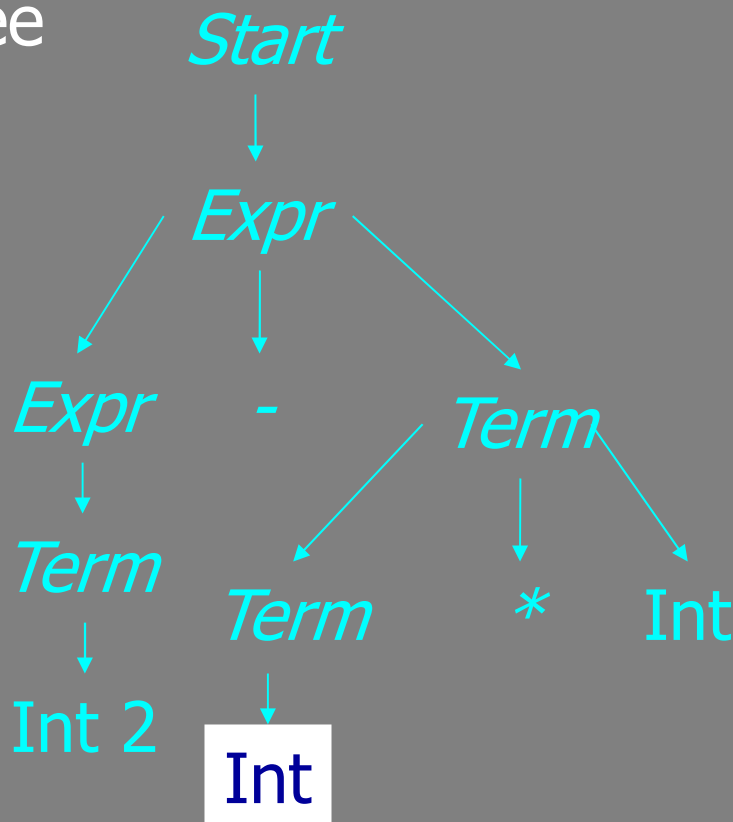
Sentential Form

2 - *Term**Int

Applied Production

Term → *Term* * Int

Parsing Example



Remaining Input

2*2

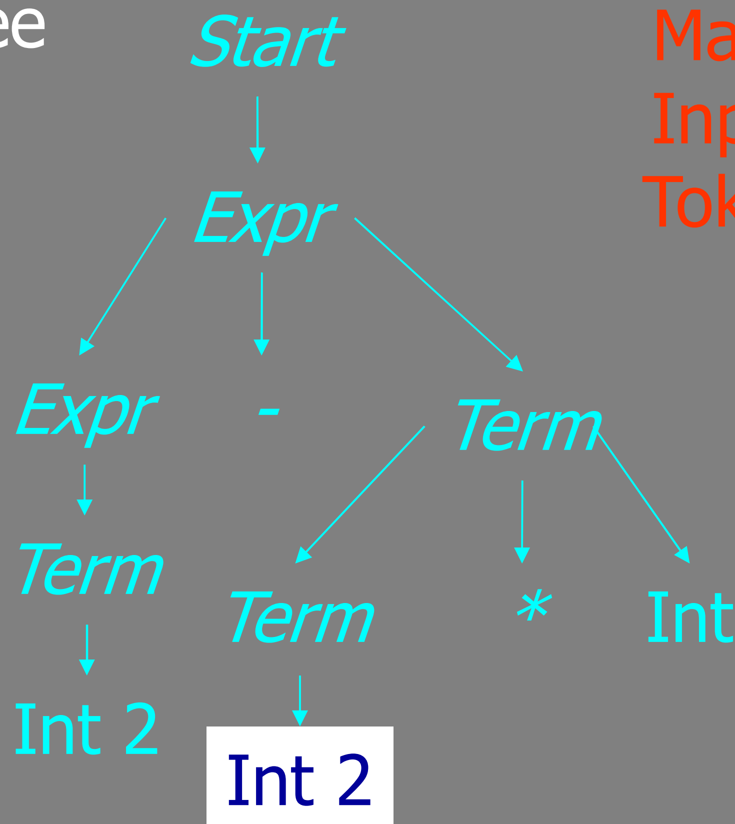
Sentential Form

2 - Int * Int

Applied Production

Term → Int

Parsing Example



Match Input Token!

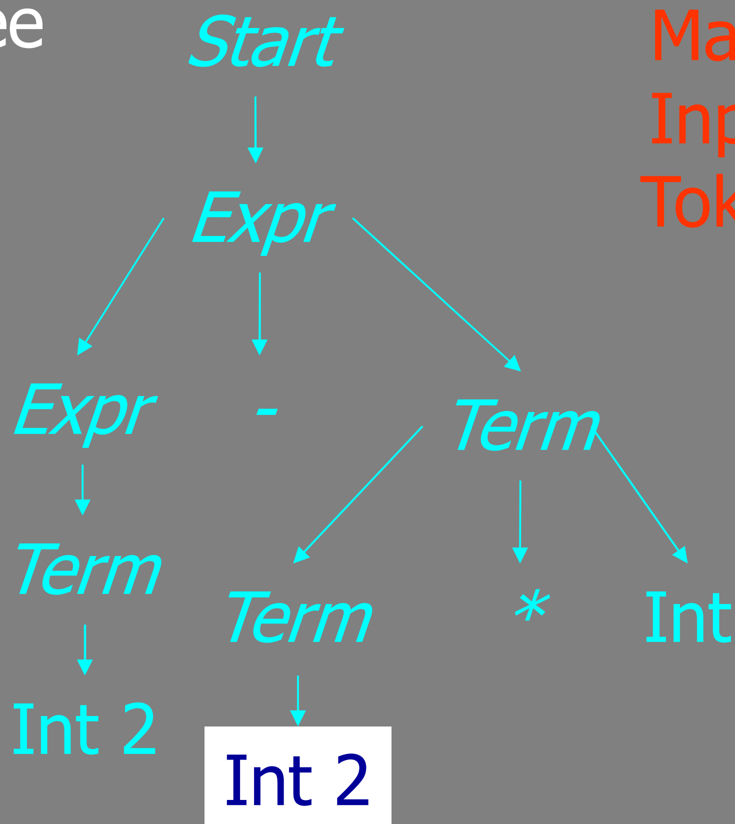
Remaining Input

2*2

Sentential Form

2 - 2 * Int

Parsing Example



Match Input Token!

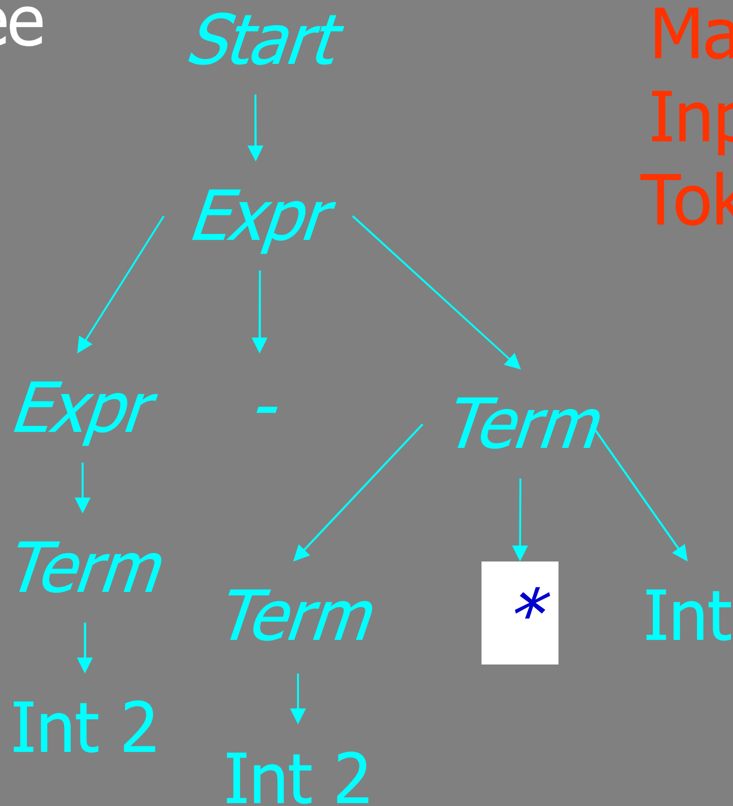
Remaining Input

*2

Sentential Form

2 - 2 * Int

Parsing Example



Match Input Token!

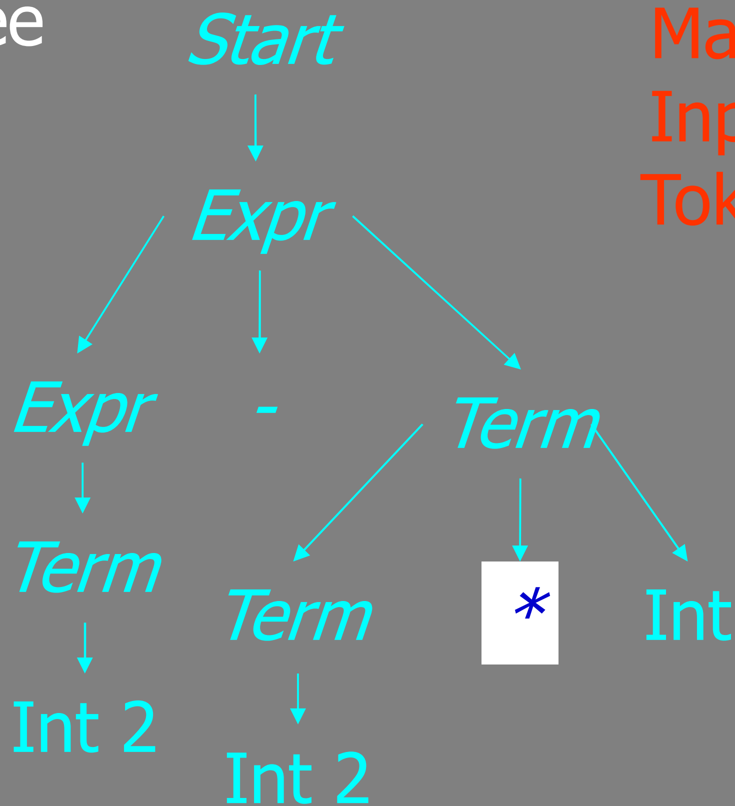
Remaining Input

*2

Sentential Form

2 - 2 * Int

Parsing Example



Match Input Token!

Remaining Input

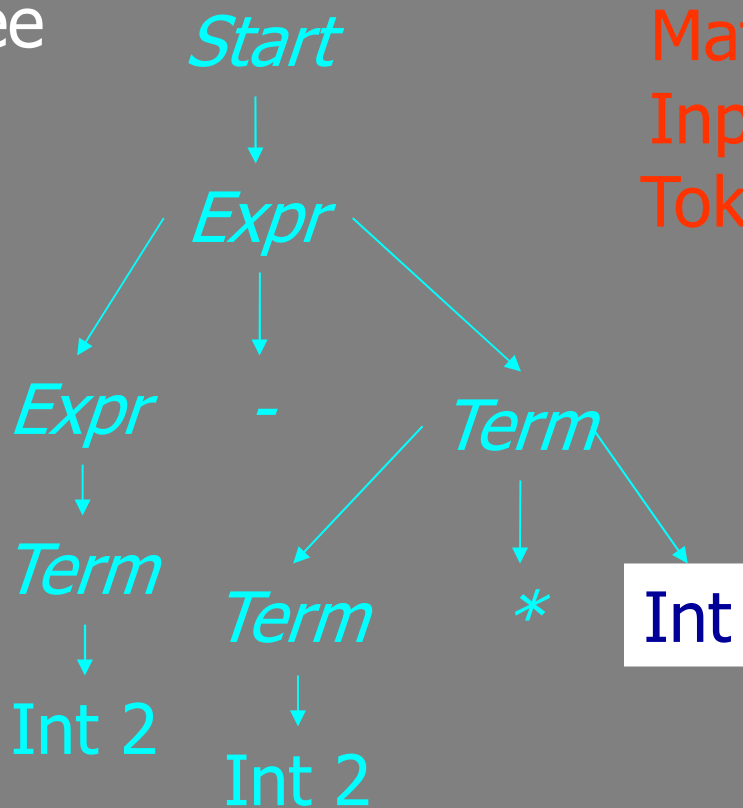
2

Sentential Form

2 - 2 * Int

Parsing Example

Parse Tree



Match Input Token!

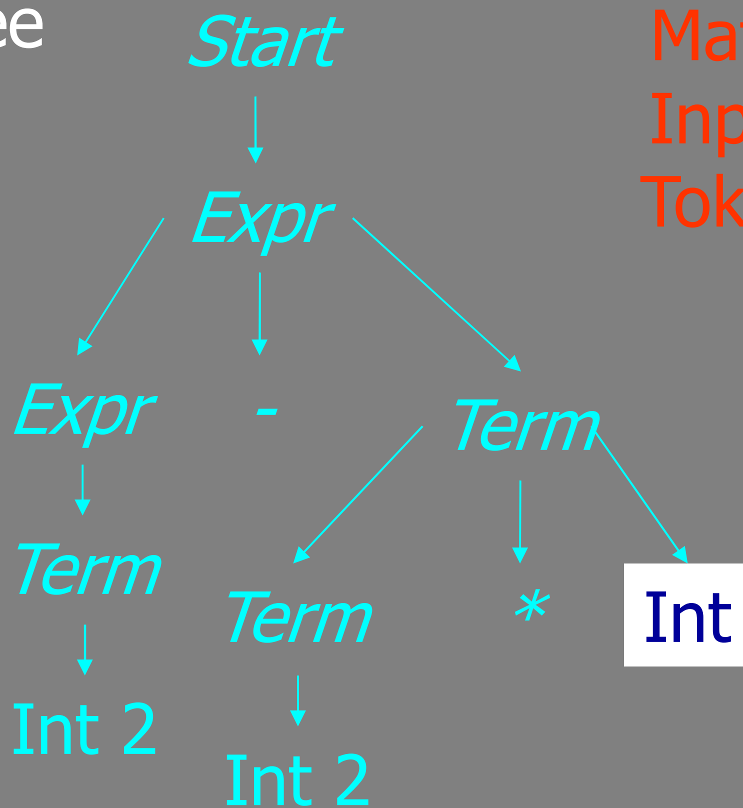
Remaining Input

2

Sentential Form

2 - 2 * Int

Parsing Example



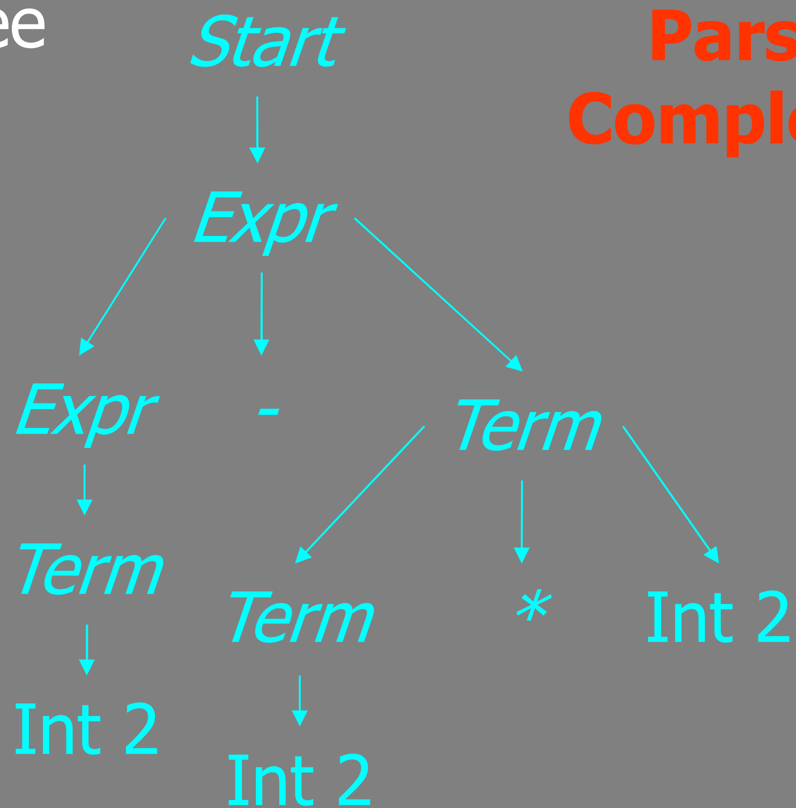
Match
Input
Token!

Remaining Input

Sentential Form

2 - 2 * Int

Parsing Example



Parse Complete!

Remaining Input

Sentential Form

2 - 2*2

Summary

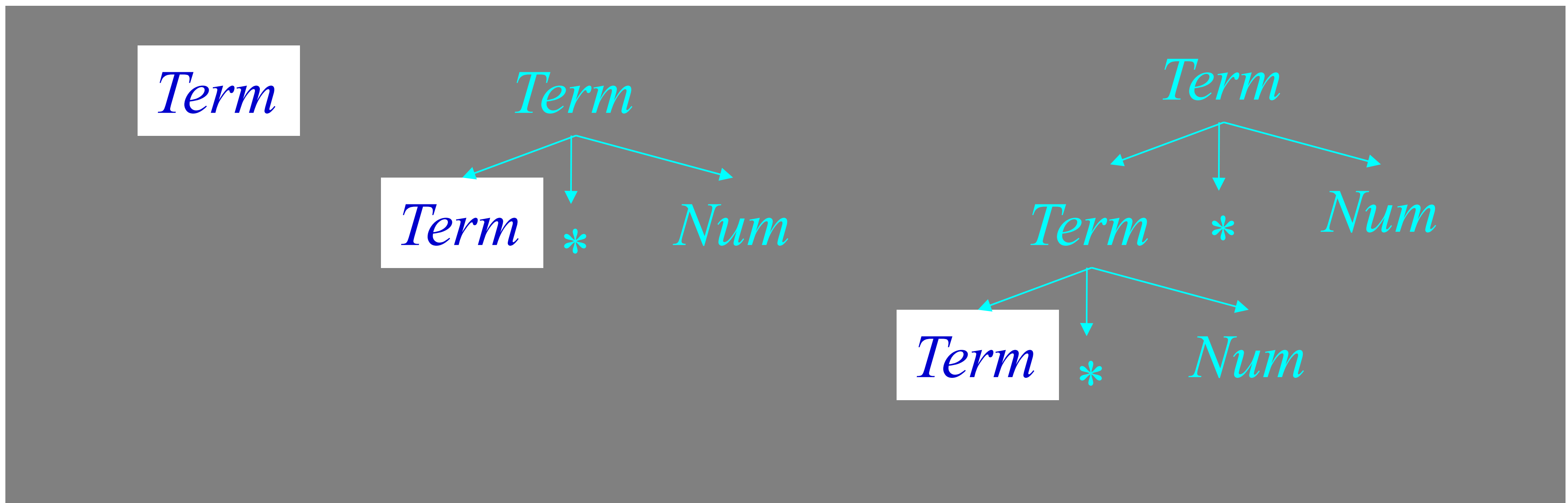
- Three Actions (Mechanisms)
  - Apply production to expand current nonterminal in parse tree
  - Match current terminal (consuming input)
  - Accept the parse as correct
- Parser generates preorder traversal of parse tree
  - visit parents before children
  - visit siblings from left to right

Policy Problem

- Which production to use for each nonterminal?
- Classical Separation of Policy and Mechanism
- One Approach: Backtracking
  - Treat it as a search problem
  - At each choice point, try next alternative
  - If it is clear that current try fails, go back to previous choice and try something different
- General technique for searching
- Used a lot in classical AI and natural language processing (parsing, speech recognition)

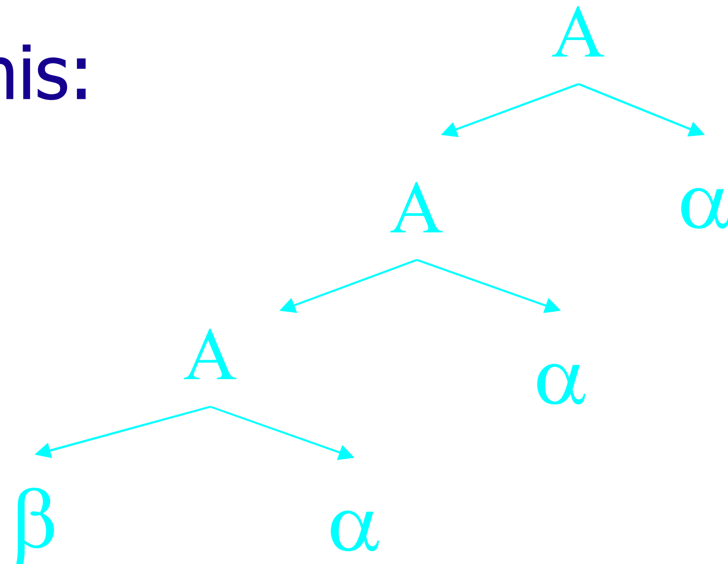
Left Recursion + Top-Down Parsing = Infinite Loop

- Example Production: $Term \rightarrow Term * Num$
- Potential parsing steps:



Eliminating Left Recursion

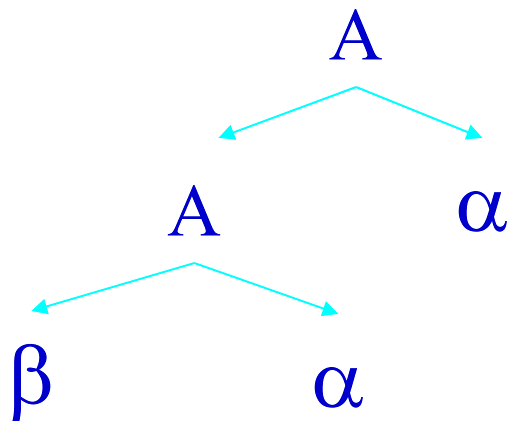
- Start with productions of form
  - $A \rightarrow A\ \alpha$
  - $A \rightarrow \beta$
  - α, β sequences of terminals and nonterminals that do not start with A
- Repeated application of $A \rightarrow A\ \alpha$ builds parse tree like this:



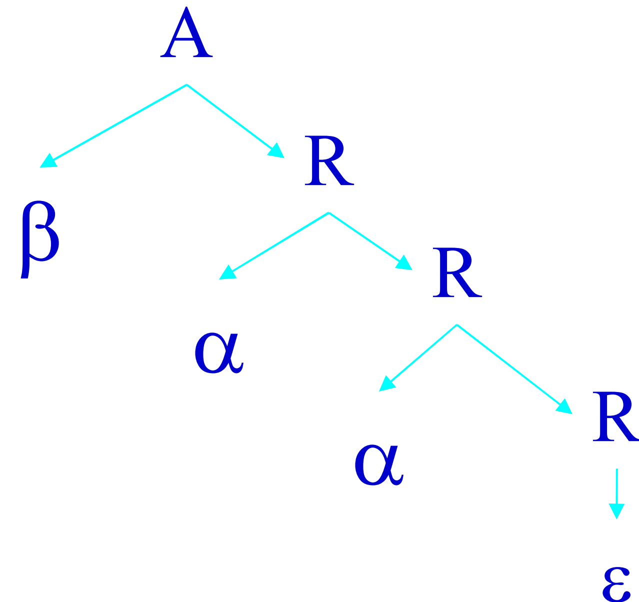
Eliminating Left Recursion

- Replacement productions
  - $A \to A\ \alpha$ $A \to \beta\ R$ R is a new nonterminal
  - $A \to \beta$ $R \to \alpha\ R$
  -     $R \to \varepsilon$

Old Parse Tree



New Parse Tree



Hacked Grammar

Original Grammar Fragment

$Term \rightarrow Term * \text{Int}$

$Term \rightarrow Term / \text{Int}$

$Term \rightarrow \text{Int}$

New Grammar Fragment

$Term \rightarrow \text{Int}\ Term'$

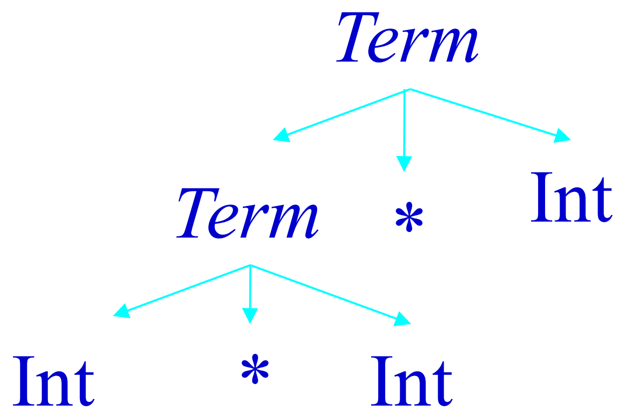
$Term' \rightarrow * \text{Int}\ Term'$

$Term' \rightarrow / \text{Int}\ Term'$

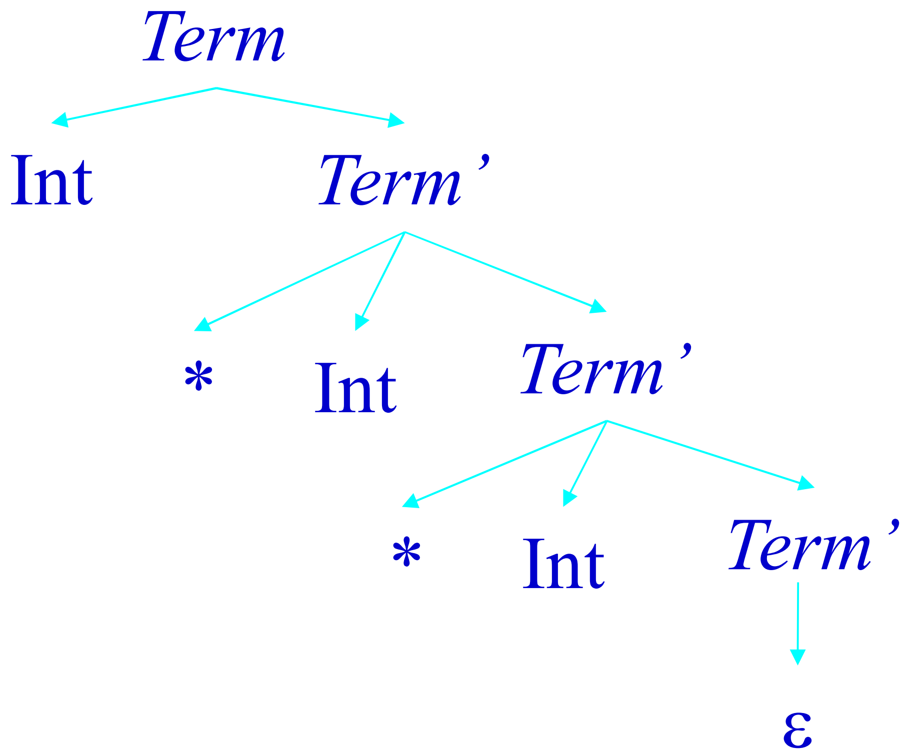
$Term' \rightarrow \varepsilon$

Parse Tree Comparisons

Original Grammar



New Grammar



Eliminating Left Recursion

- Changes search space exploration algorithm
  - Eliminates direct infinite recursion
  - But grammar less intuitive
- Sets things up for predictive parsing

Predictive Parsing

- Alternative to backtracking
- Useful for programming languages, which can be designed to make parsing easier
- Basic idea
  - Look ahead in input stream
  - Decide which production to apply based on next tokens in input stream
  - We will use one token of lookahead

Predictive Parsing Example Grammar

$Start \rightarrow Expr$

$Expr \rightarrow Term\ Expr'$

$Expr' \rightarrow +\ Expr'$

$Expr' \rightarrow -\ Expr'$

$Expr' \rightarrow \varepsilon$

$Term \rightarrow \text{Int}\ Term'$

$Term' \rightarrow *\ \text{Int}\ Term'$

$Term' \rightarrow /\ \text{Int}\ Term'$

$Term' \rightarrow \varepsilon$

Choice Points

- Assume *Term′* is current position in parse tree
- Have three possible productions to apply

  Term′ → *** Int *Term′*

  Term′ → */* Int *Term′*

  Term′ → ε

- Use next token to decide
  - If next token is *, apply *Term′* → *** Int *Term′*
  - If next token is /, apply *Term′* → */* Int *Term′*
  - Otherwise, apply *Term′* → ε

Predictive Parsing + Hand Coding = Recursive Descent Parser

- One procedure per nonterminal *NT*
  - Productions $NT \to \beta_1$, …, $NT \to \beta_n$
  - Procedure examines the current input symbol T to determine which production to apply
    - If $T \in First(\beta_k)$
    - Apply production k
    - Consume terminals in β_k (check for correct terminal)
    - Recursively call procedures for nonterminals in β_k
  - Current input symbol stored in global variable token
- Procedures return
  - true if parse succeeds
  - false if parse fails

Bottom Up Parsing