Compilation

0368-3133

Lecture 2:

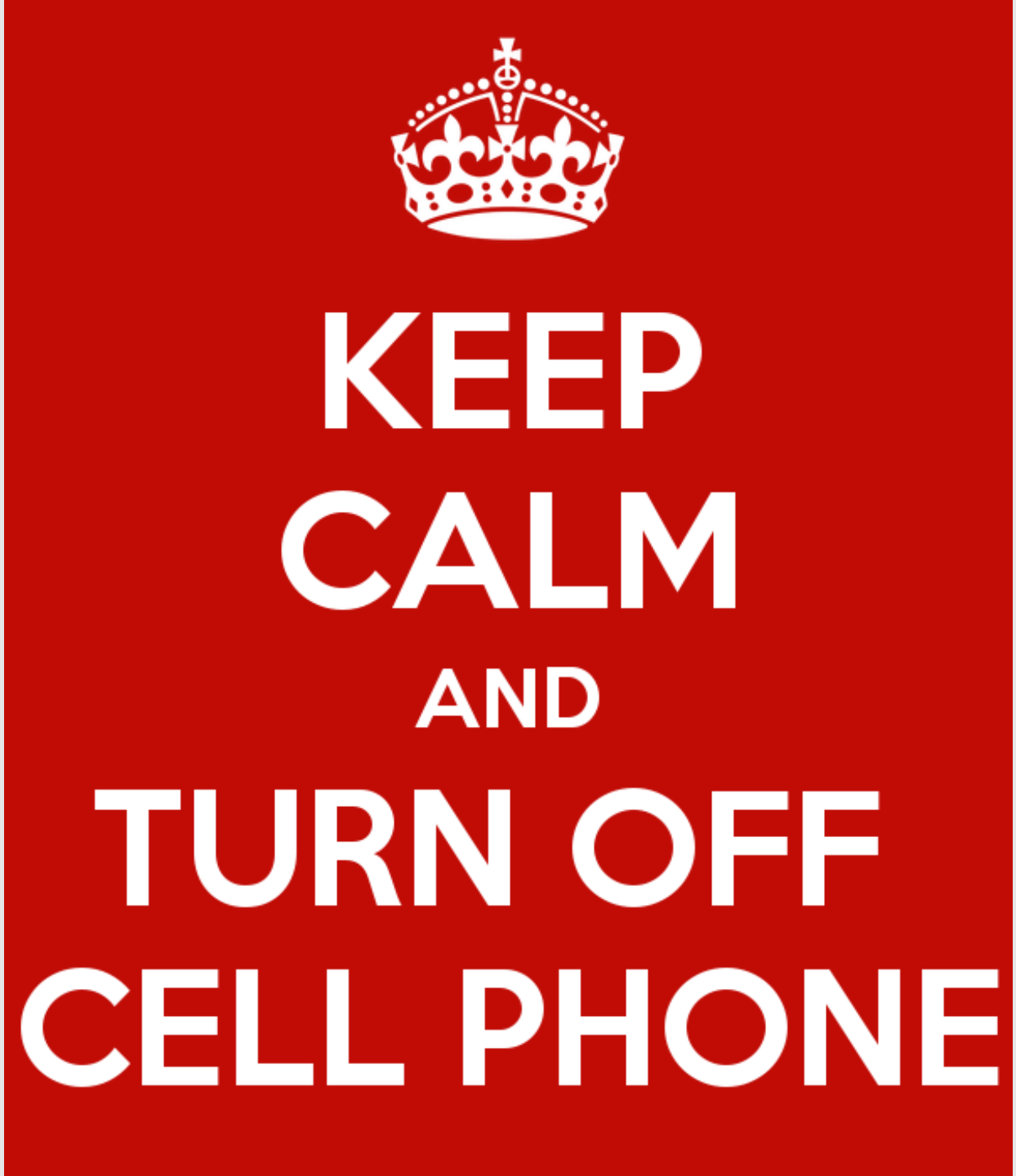
Lexical Analysis

Syntax Analysis: CFLs, PDAs,

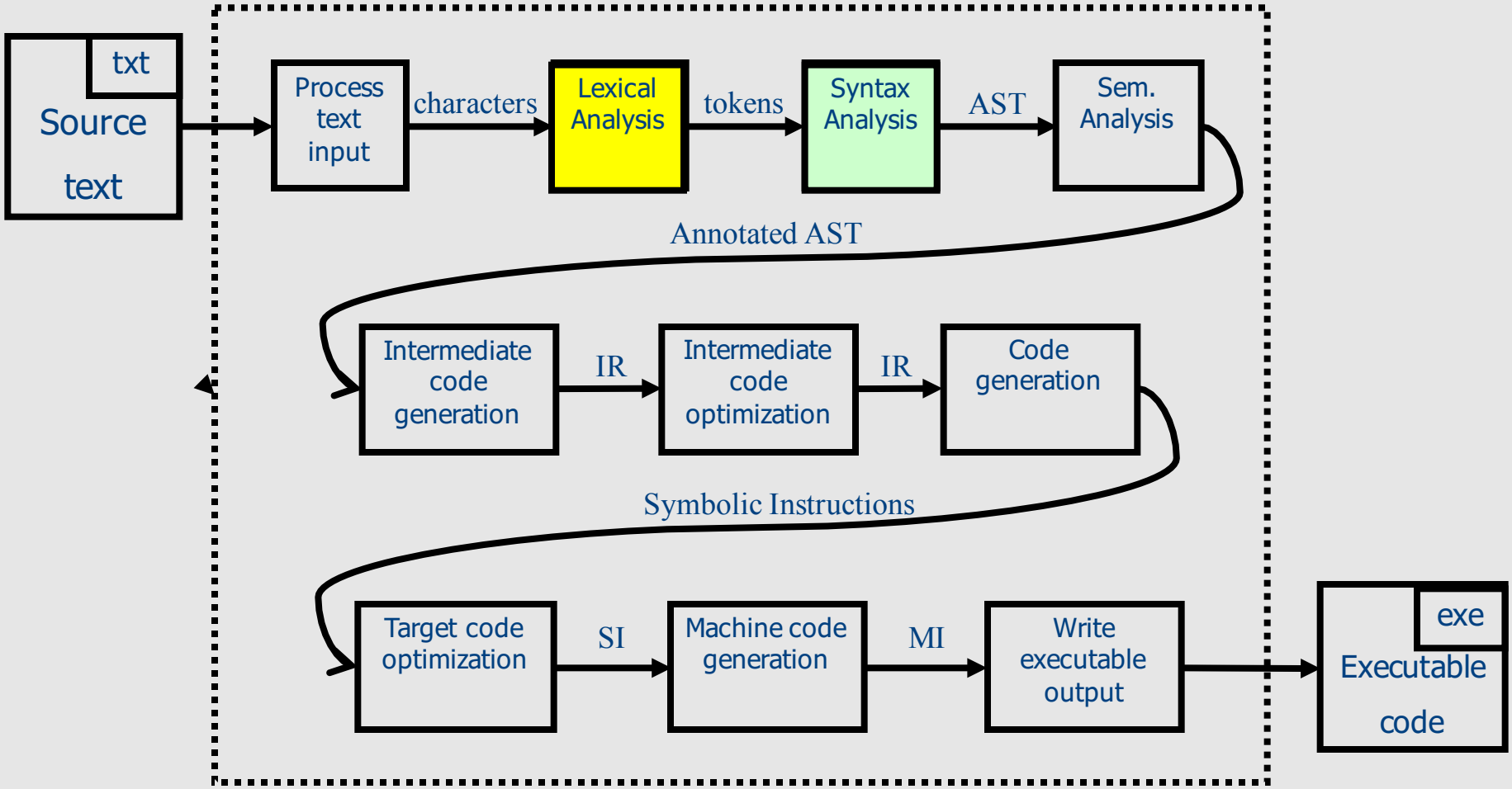
Top Down parsing

Lectures begin at 09:10, 10:10, 11:10

Noam Rinetzky


KEEP
CALM
AND
TURN OFF
CELL PHONE

The Real Anatomy of a Compiler



Lexical Analysis: What

- Input: program text (file)
- Output: sequence of tokens

What does Lexical Analysis do?

- Scan the input
- Partitions the text into stream of tokens
  - Numbers
  - Identifiers
  - Keywords
  - Punctuation
- Tokens usually represented as (kind, value)
- Defined using regular expressions*

- "word" in the source language
- "meaningful" to the syntactical analysis

What does Lexical Analysis do?

- Language: fully parenthesized expressions

Context free language

Expr → Num | LP Expr Op Expr RP

Regular languages

Num → Dig | Dig Num

Dig → ‘0’ | ‘1’ | ‘2’ | ‘3’ | ‘4’ | ‘5’ | ‘6’ | ‘7’ | ‘8’ | ‘9’

LP → ‘(’

RP → ‘)’

Op → ‘+’ | ‘*’

((23 + 7) * 19)

Lexical Analysis: How

- Define tokens using regular expressions

- Construct a nondeterministic finite-state automaton (NFA) from regular expression

- Determinize the NFA into a deterministic finite-state automaton (DFA)

- DFA can be directly used to identify tokens

What does Lexical Analysis do?

- Language: fully parenthesized expressions

Context free language

Expr → Num | LP Expr Op Expr RP

Regular languages

Num → Dig | Dig Num

Dig → ‘0’ | ‘1’ | ‘2’ | ‘3’ | ‘4’ | ‘5’ | ‘6’ | ‘7’ | ‘8’ | ‘9’

LP → ‘(’

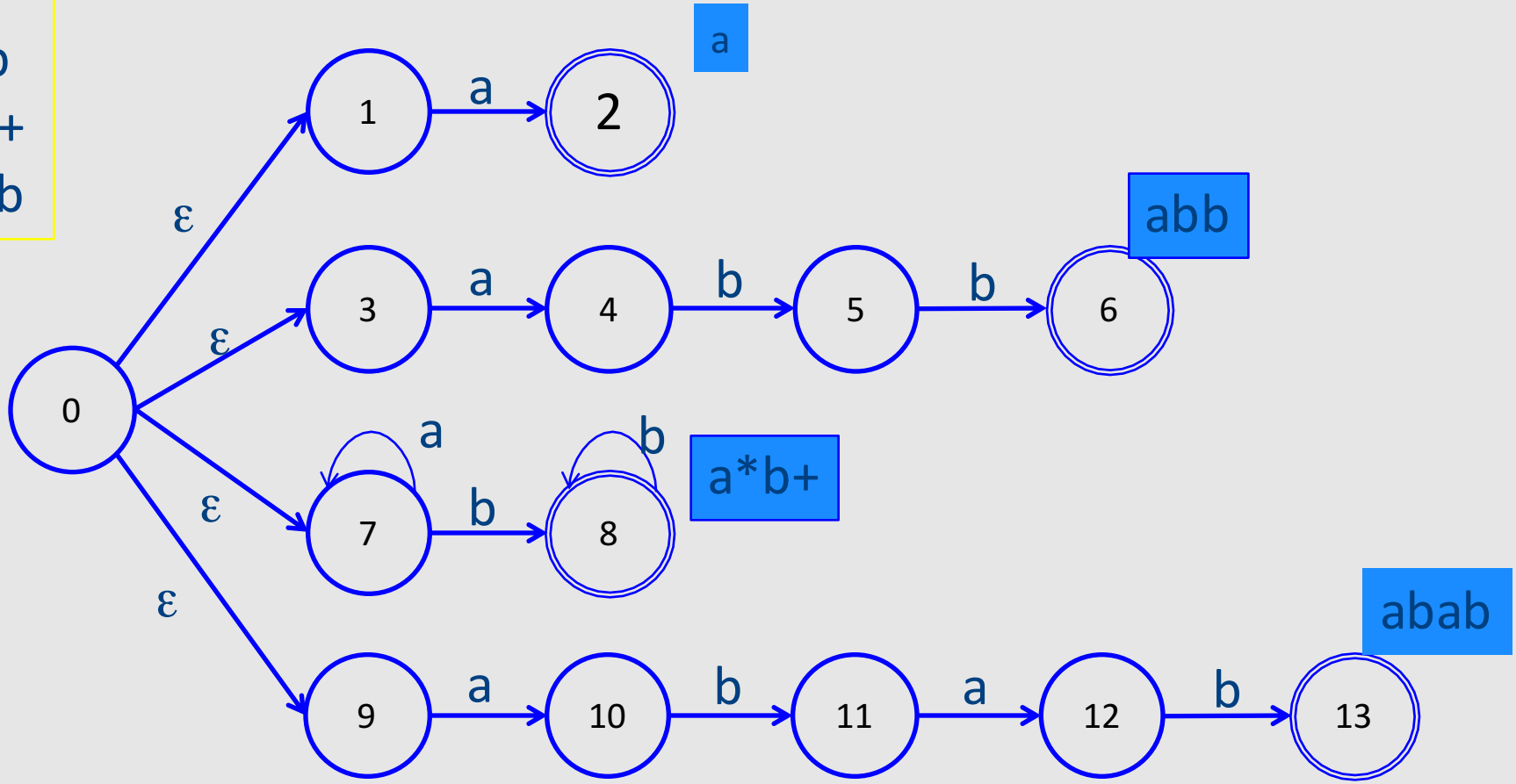
RP → ‘)’

Op → ‘+’ | ‘*’

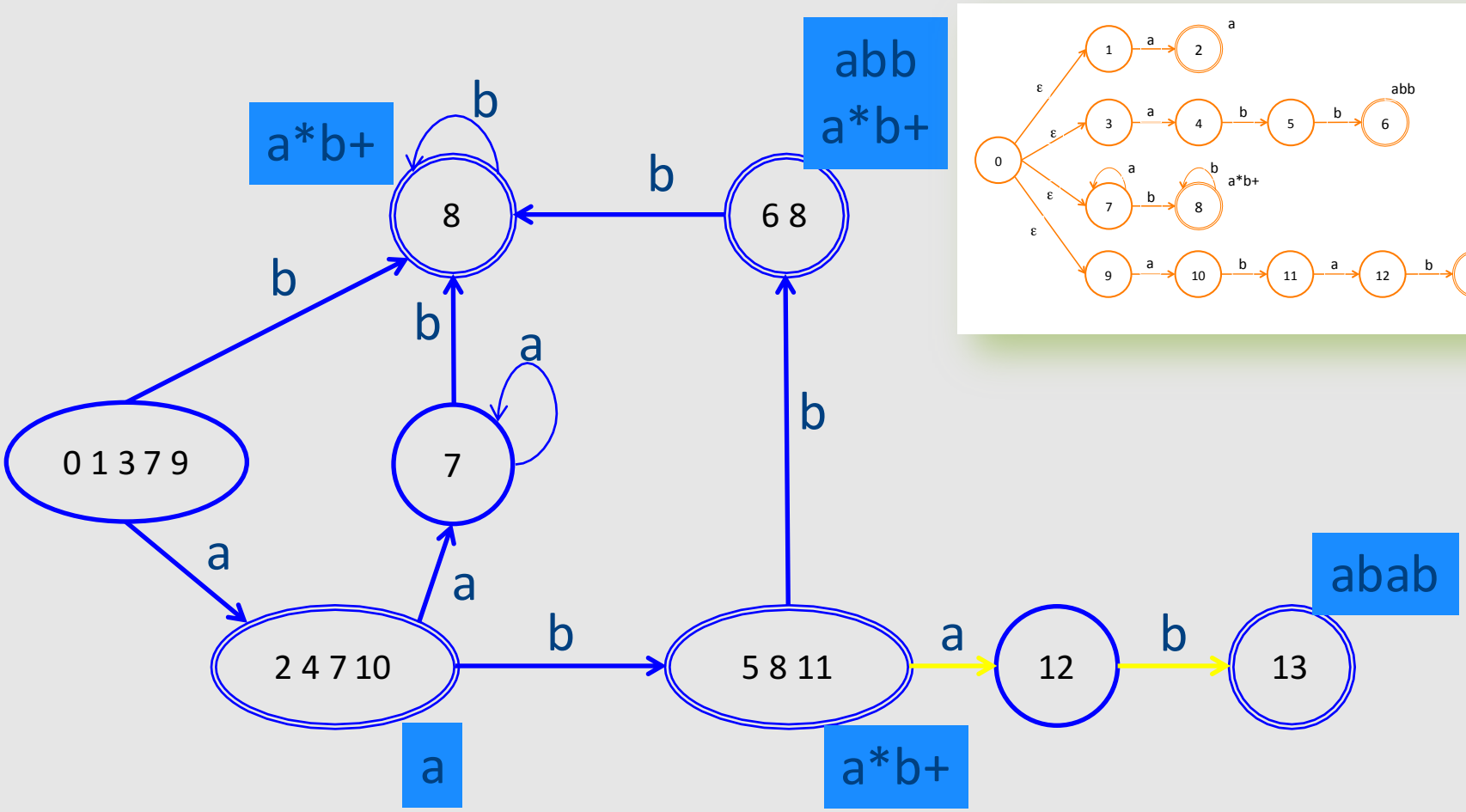
Actually, we combine automata

combines

a
abb
a*b+
abab



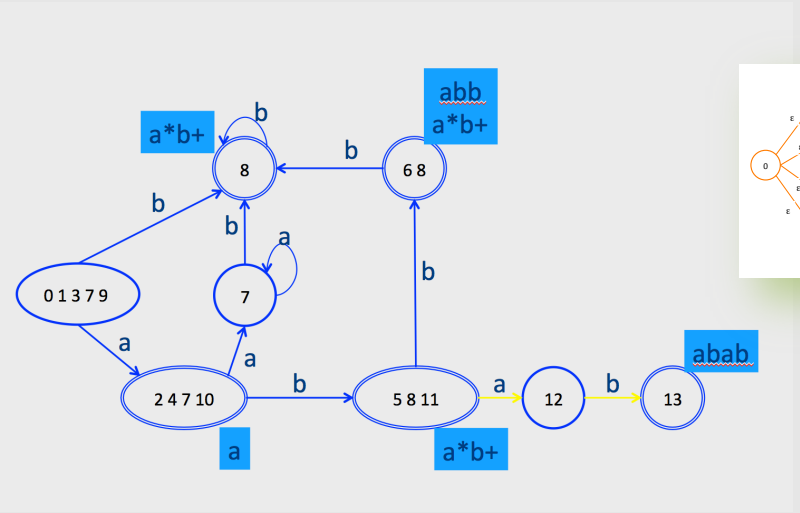
Corresponding DFA

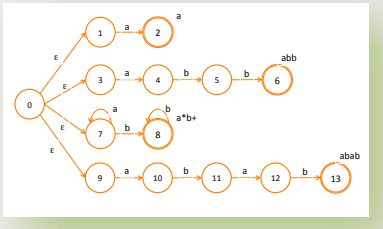


Scanning with DFA

- Run until stuck
  - **Remember last accepting state**
- Go back to accepting state
- Return token

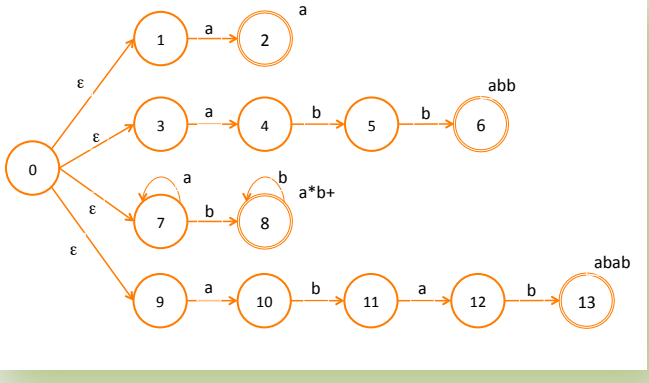
Ambiguity resolution





- Longest word
- Tie-breaker based on **order of rules** when words have same length

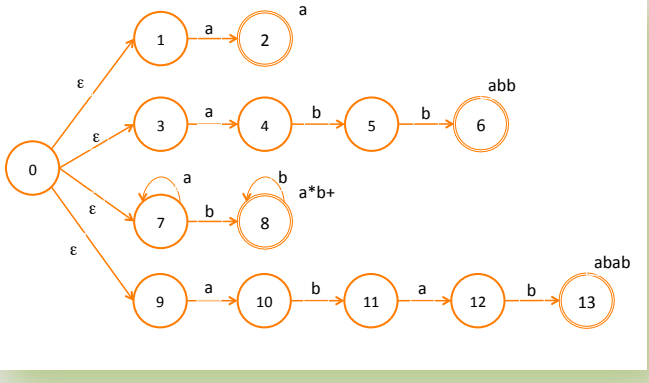
Examples



abaa: gets stuck after aba in state 12, backs up to state (5 8 11) pattern is a*b+, token is ab

Tokens: <a*b+, ab> <a,a><a,a>

Examples



abba: stops after second b in (6 8), token is abb because it comes first in spec

Tokens: <abb, abb> <a,a>

Summary of Construction

- Describe tokens as regular expressions
  - Decide attributes (values) to save for each token

- Regular expressions turned into a DFA
  - Also, records which attributes (values) to keep

- Lexical analyzer simulates the run of an automata with the given transition table on any input string

A Few Remarks

- Turning an NFA to a DFA is expensive, but
  - Exponential in the worst case
  - In practice, works fine
- The construction is done once per-language
  - At Compiler construction time
  - Not at compilation time

Implementation

Implementation by Example

if	`if`	`{ return IF; }`
xy, i, zs98	`[a-z][a-z0-9]*`	`{ return ID; }`
3,32, 032	`[0-9]+`	`{ return NUM; }`
0.55, 33.1	`[0-9]"."[0-9]+|[0-9]*"."[0-9]+`	`{ return REAL; }`
--comm\n \n, \t, " "	`(\-\-[a-z]*\n)|(" "|\n|\t)`	`{ ; }`
	`.`	`{ error(); }`

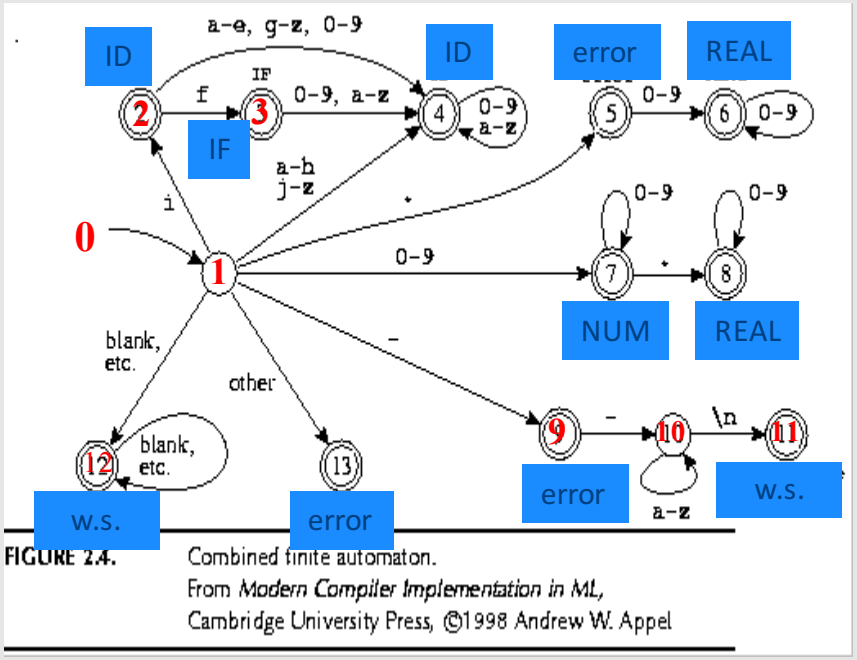
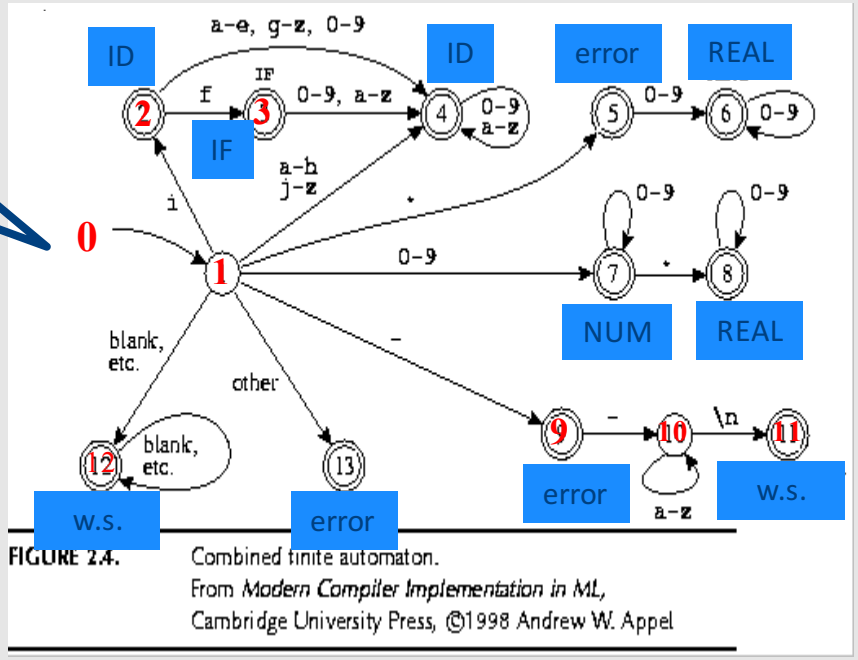


FIGURE 2.4. Combined finite automaton.
From *Modern Compiler Implementation in ML*,
Cambridge University Press, ©1998 Andrew W. Appel

State 0 is "stuck" / "dead"




```
int edges[][256]= {
                    /* …, 0, 1, 2, 3, ..., -, e, f, g, h, i, j,  ... */
/* state 0  */   {0, …, 0, 0, …,  0, 0, 0, 0, 0, ..., 0, 0, 0, 0, 0, 0},
/* state 1  */   {13, … , 7, 7, 7, 7,  …,  9,  4, 4, 4, 4, 2, 4, …, 13, 13},
/* state 2  */   {0,  …, 4, 4, 4, 4,  ..., 0,  4, 3, 4, 4, 4, 4, …, 0, 0},
/* state 3  */   {0, …,  4, 4, 4, 4,  …, 0,  4, 4, 4, 4, 4, 4, , 0, 0},
/* state 4  */   {0, …,  4, 4, 4, 4,   …, 0,  4, 4, 4, 4, 4, 4, …, 0, 0},
/* state 5  */   {0, …,  6, 6, 6, 6,   …, 0,  0, 0, 0, 0, 0, 0, …,  0, 0},
/* state 6  */   {0,  …,  6, 6, 6, 6,  …, 0,  0, 0, 0, 0, 0, 0, ..., 0, 0},
/* state 7  */
/* state …  */                        ...
/* state 13 */   {0,  …,  0, 0, 0, 0,  …, 0,  0, 0, 0, 0, 0, 0, …,  0, 0}
};
```

Pseudo Code for Scanner

```
char* input = … ;

Token nextToken() {
    lastFinal = 0;
    currentState = 1 ;
    inputPositionAtLastFinal = input;
    currentPosition = input;
    while (not(isDead(currentState))) {
        nextState = edges[currentState][*currentPosition];
        if  (isFinal(nextState)) {
                lastFinal = nextState ;
                inputPositionAtLastFinal = currentPosition;
        }
        currentState = nextState;
        advance currentPosition;
    }
    input = inputPositionAtLastFinal + 1;
    return action[lastFinal];
}
```

Example

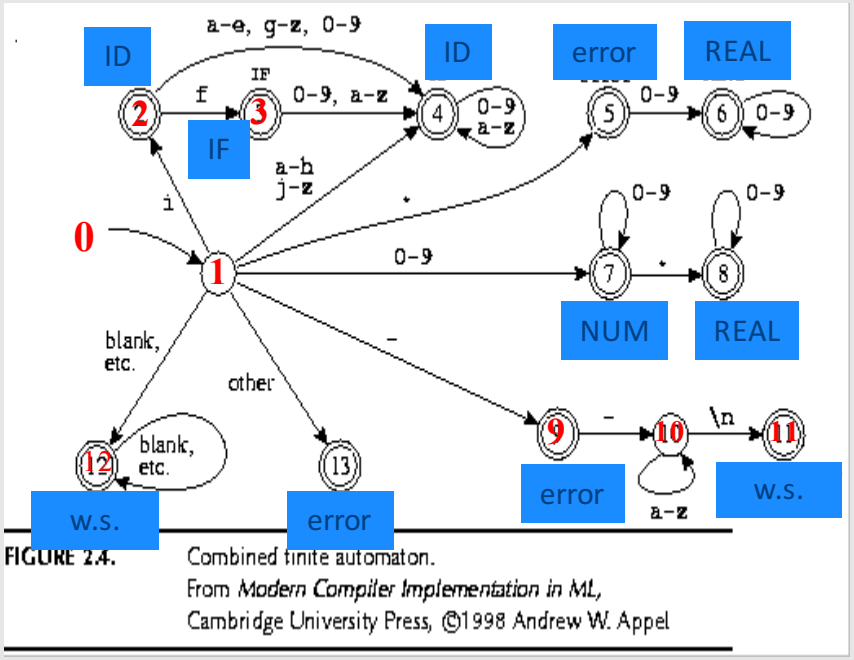


FIGURE 2.4. Combined finite automaton.
From *Modern Compiler Implementation in ML*,
Cambridge University Press, ©1998 Andrew W. Appel

Input: "if --not-a-com"

2 blanks

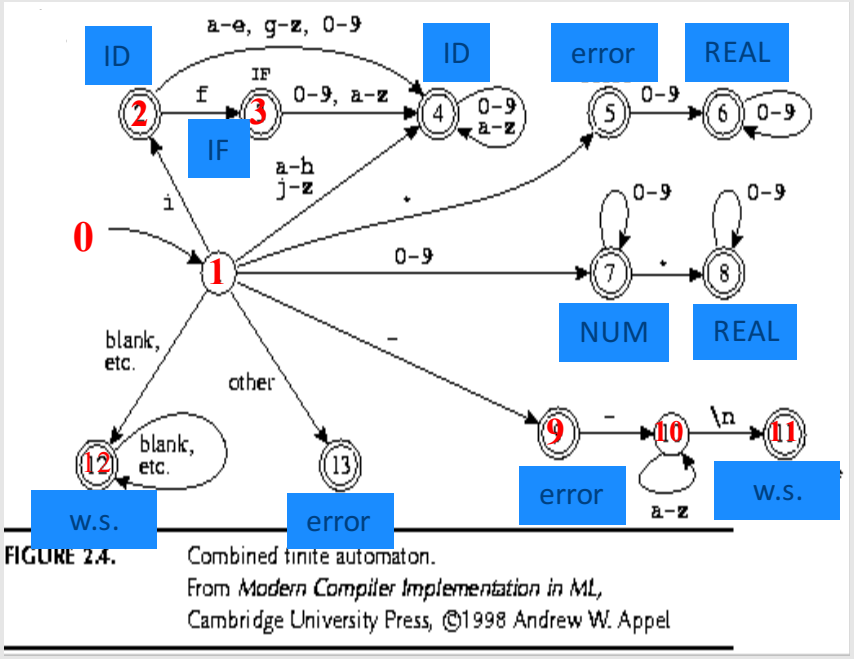


FIGURE 2.4. Combined finite automaton.
From *Modern Compiler Implementation in ML*,
Cambridge University Press, ©1998 Andrew W. Appel

return IF

final	state	input
0	1	if --not-a-com
2	2	if --not-a-com
3	3	if --not-a-com
3	0	if --not-a-com

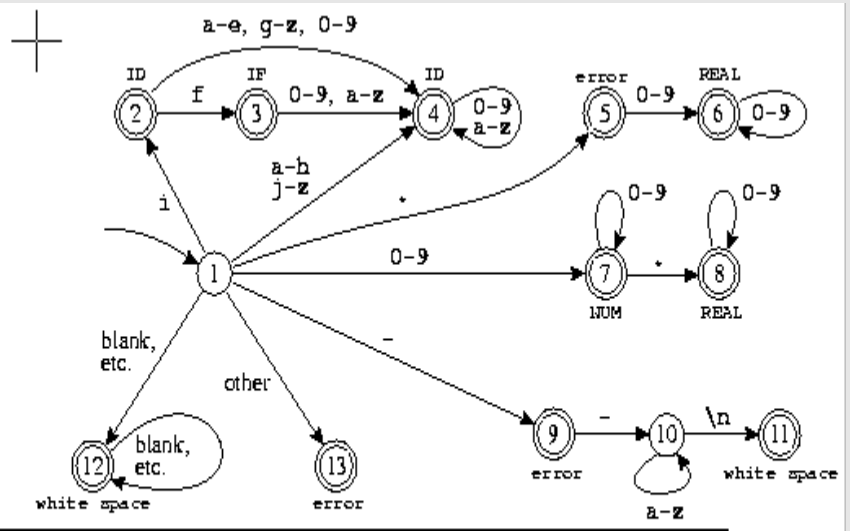


FIGURE 2.4. Combined finite automaton.
From *Modern Compiler Implementation in ML*,
Cambridge University Press, ©1998 Andrew W. Appel

found whitespace

final	state	input
0	1	--not-a-com
12	12	--not-a-com
12	12	--not-a-com
12	0	--not-a-com

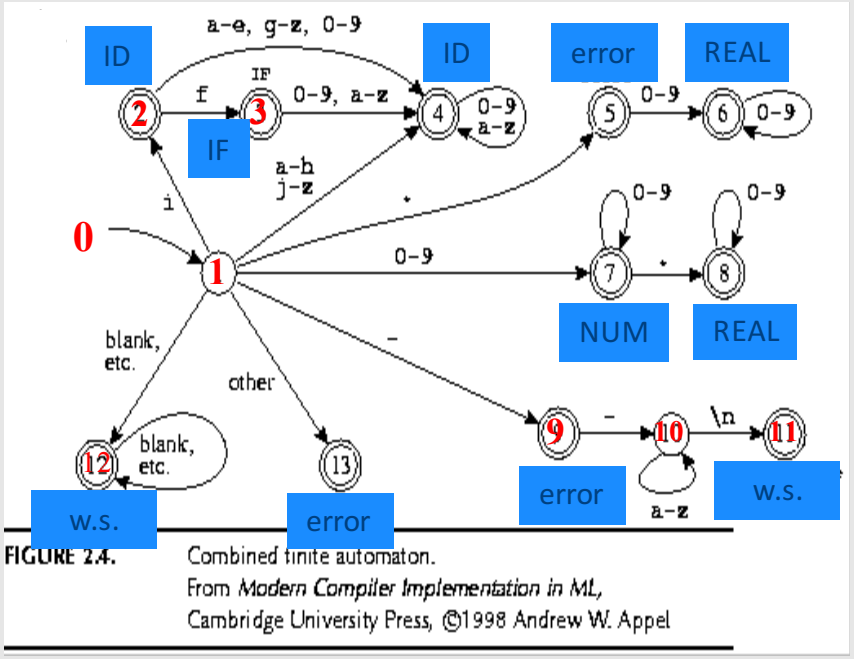


FIGURE 2.4. Combined finite automaton.
From *Modern Compiler Implementation in ML*,
Cambridge University Press, ©1998 Andrew W. Appel

error

final	state	input
0	1	--not-a-com
9	9	--not-a-com
9	10	--not-a-com
9	10	--not-a-com
9	10	--not-a-com
9	0	--not-a-com

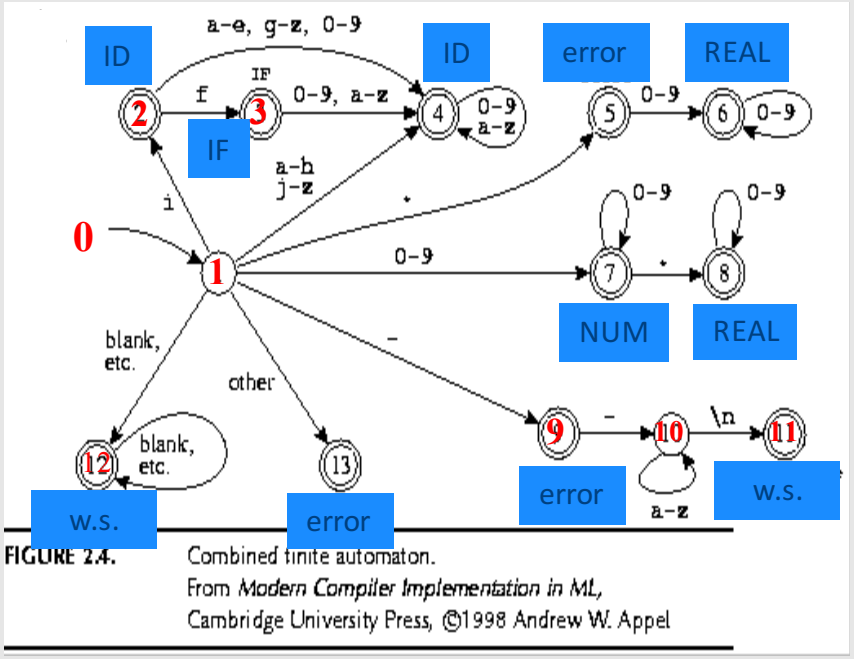


FIGURE 2.4. Combined finite automaton.
From *Modern Compiler Implementation in ML*,
Cambridge University Press, ©1998 Andrew W. Appel

error

final	state	input
0	1	-not-a-com
9	9	-not-a-com
9	0	not-a-com
9	0	not-a-com
9	0	not-a-com

Concluding remarks

- Efficient scanner
- Minimization
- Error handling
- Automatic creation of lexical analyzers

Efficient Scanners

- Efficient state representation
- Input buffering
- Using switch and gotos instead of tables

Minimization

- Create a non-deterministic automaton (NDFA) from every regular expression
- Merge all the automata using epsilon moves (like the | construction)
- Construct a deterministic finite automaton (DFA)
  - State priority
- Minimize the automaton
  - separate accepting states by token kinds

Example

```
if                  { return IF; }
[a-z][a-z0-9]*      { return ID; }
[0-9]+              { return NUM; }
```

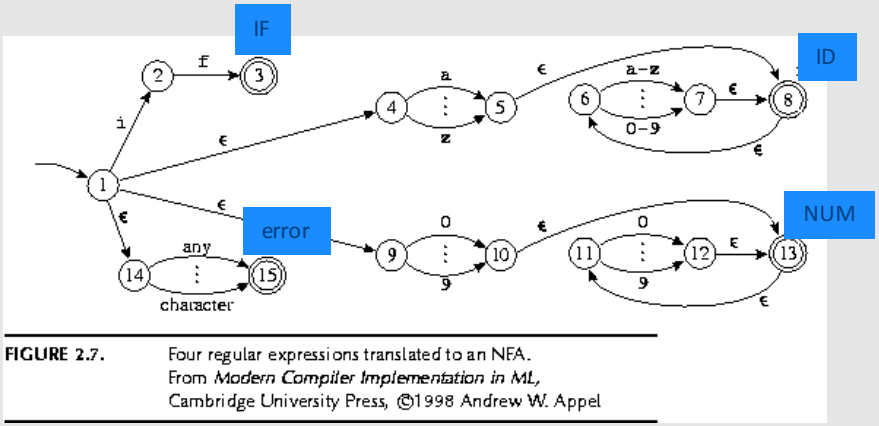


FIGURE 2.7. Four regular expressions translated to an NFA.
From *Modern Compiler Implementation in ML*,
Cambridge University Press, ©1998 Andrew W. Appel

Modern compiler implementation in ML, Andrew Appel, (c)1998, Figures 2.7,2.8

Example

```
if                  { return IF; }
[a-z][a-z0-9]*      { return ID; }
[0-9]+              { return NUM; }
```

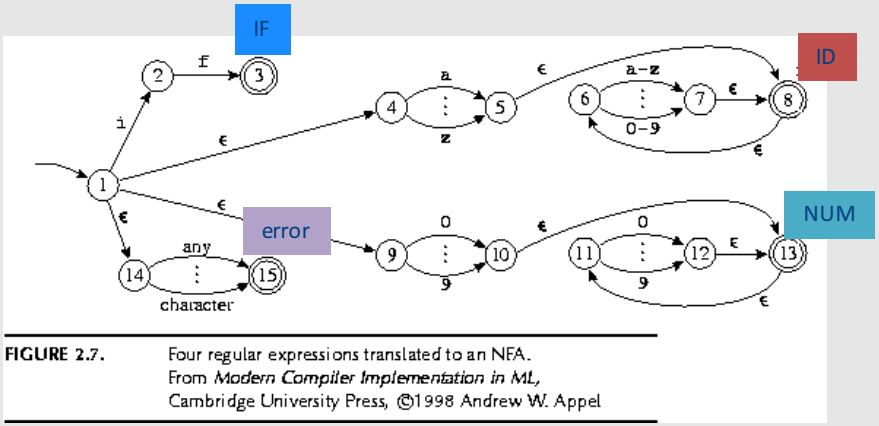
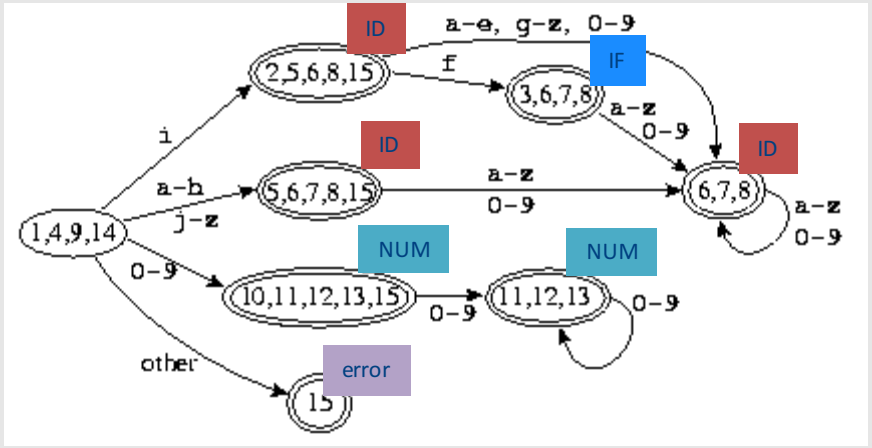
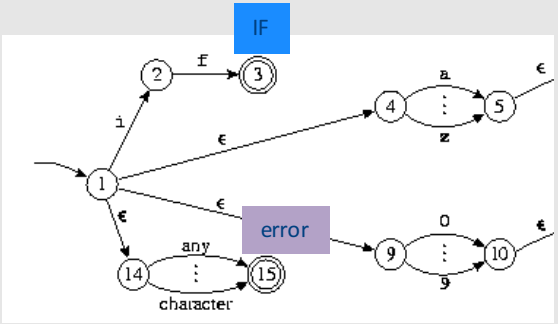


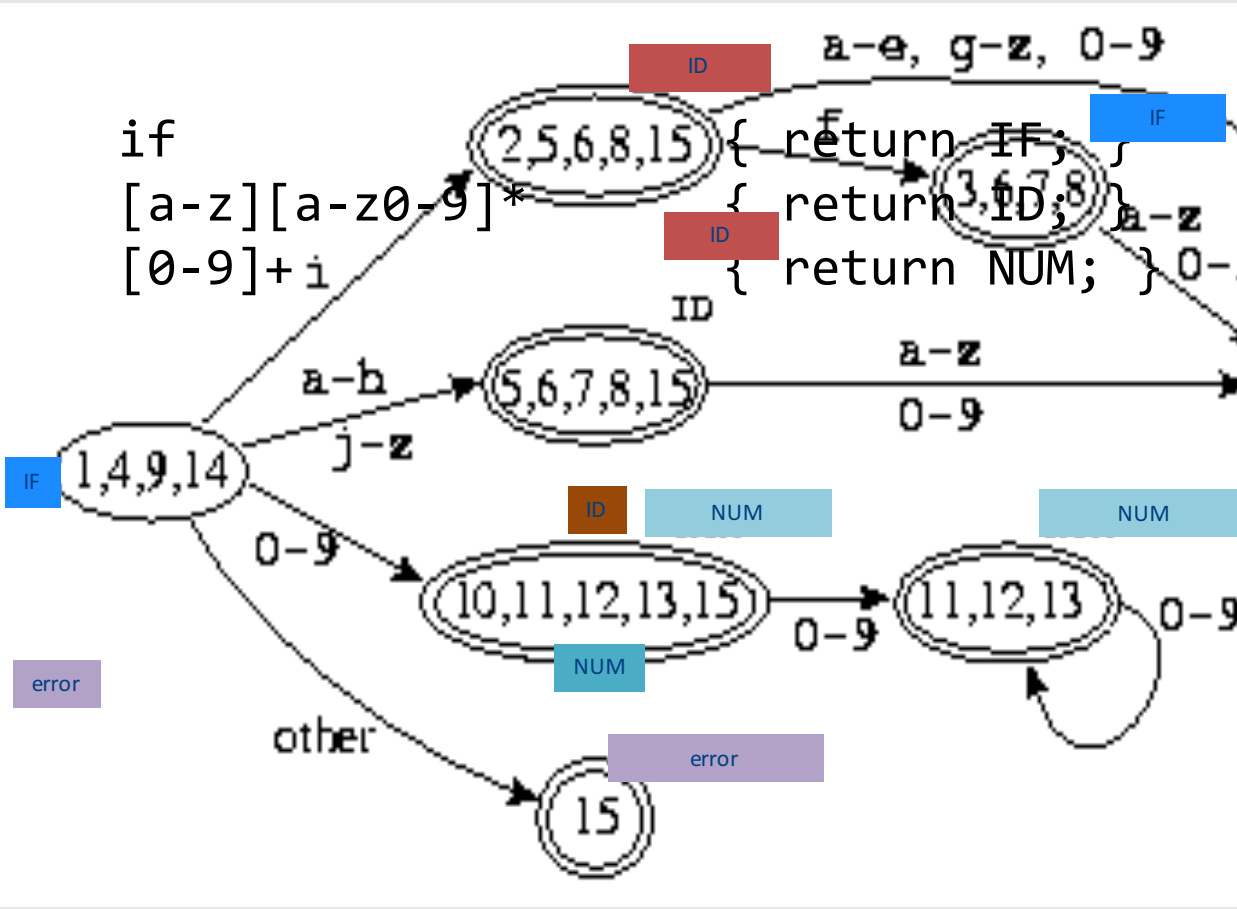
FIGURE 2.7. Four regular expressions translated to an NFA.
From *Modern Compiler Implementation in ML*,
Cambridge University Press, ©1998 Andrew W. Appel



Modern compiler implementation in ML, Andrew Appel, (c)1998, Figures 2.7,2.8

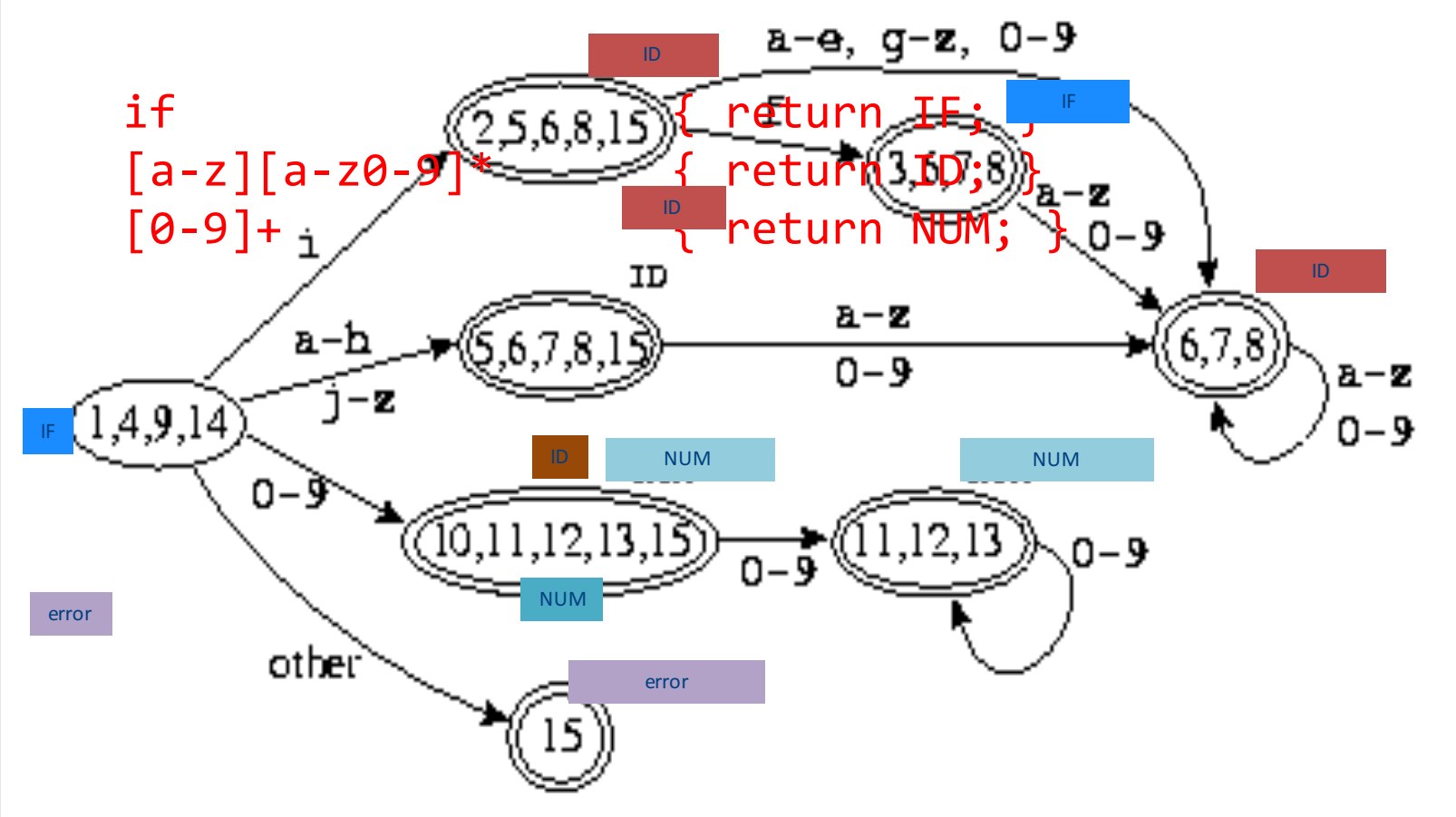
Example





Modern compiler implementation in ML, Andrew Appel, (c)1998, Figures 2.7,2.8

Example



Modern compiler implementation in ML, Andrew Appel, (c)1998, Figures 2.7,2.8

Error Handling

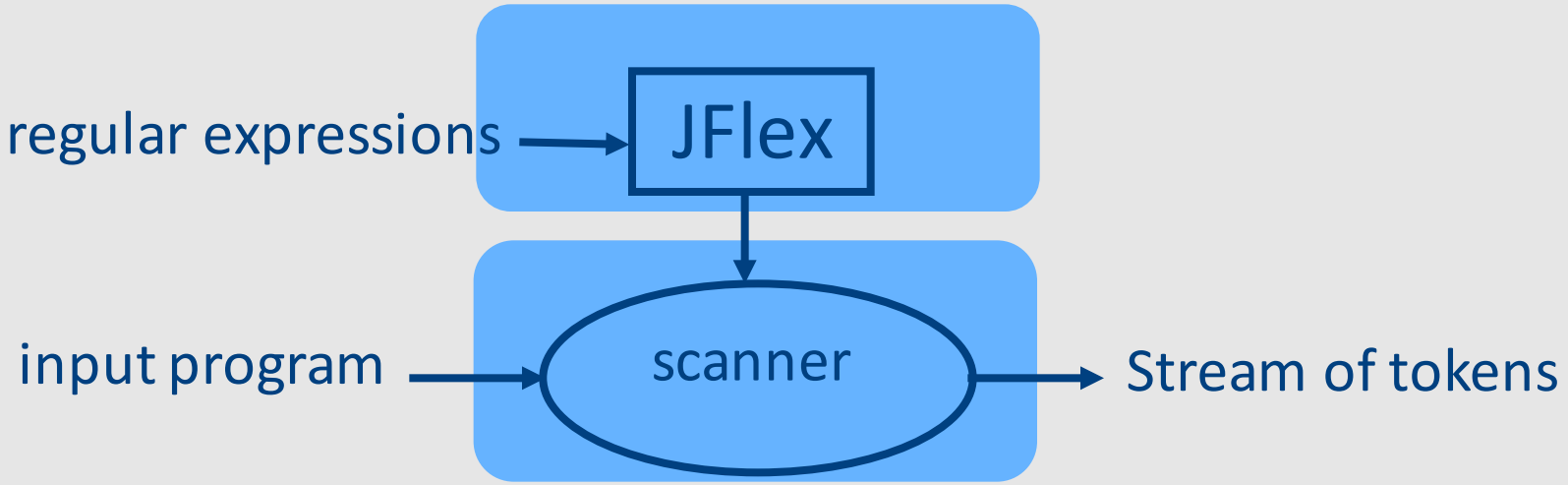
- Many errors cannot be identified at this stage
- Example: “fi (a==f(x))”. Should “fi” be “if”? Or is it a routine name?
  - We will discover this later in the analysis
  - At this point, we just create an identifier token

- Sometimes the lexeme does not match any pattern
  - Easiest: eliminate letters until the beginning of a legitimate lexeme
  - Alternatives: eliminate/add/replace one letter, replace order of two adjacent letters, etc.

- Goal: allow the compilation to continue
- Problem: errors that spread all over

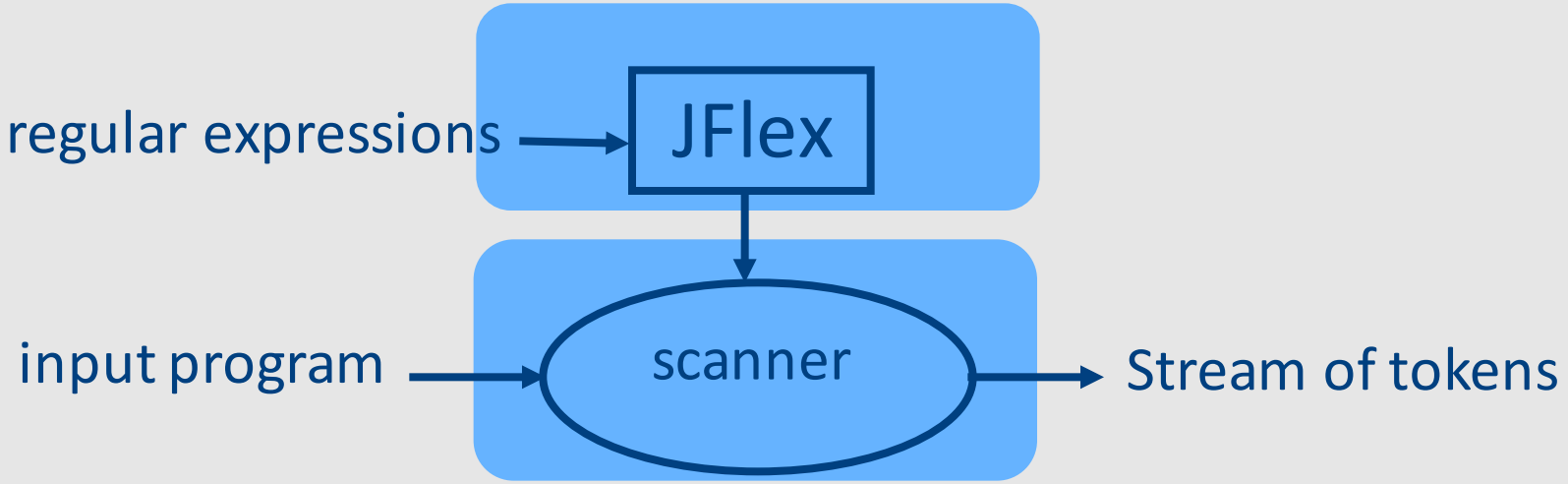
Automatically generated scanners

- Use of Program-Generating Tools
  - Specification ➔ Part of compiler
  - Compiler-Compiler



Use of Program-Generating Tools

- Input: regular expressions and actions
  - Action = Java code
- Output: a scanner program that
  - Produces a stream of tokens
  - Invoke actions when pattern is matched



Line Counting Example

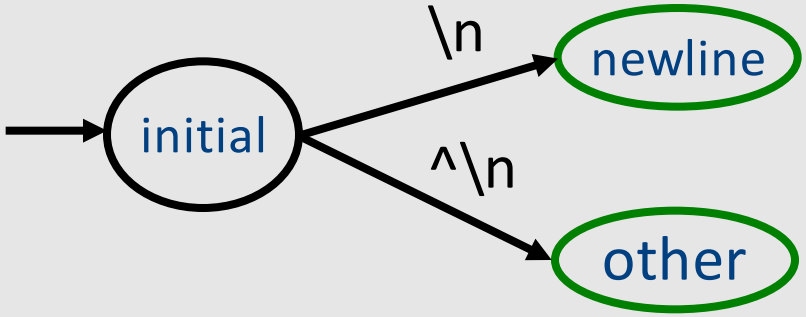
- Create a program that counts the number of lines in a given input text file

Creating a Scanner using Flex

```
int num_lines = 0;
%%
\n      ++num_lines;
.       ;
%%
main() {
  yylex();
  printf( "# of lines = %d\n", num_lines);
}
```

Creating a Scanner using Flex

```
int num_lines = 0;
%%
\n      ++num_lines;
.       ;
%%
main() {
  yylex();
  printf( "# of lines = %d\n", num_lines);
}
```



JFLex Spec File

User code: Copied directly to Java file

%%

JFlex directives: macros, state names

%%

Lexical analysis rules:

- Optional state, regular expression, action
- How to break input to tokens
- Action when token matched

Possible source of javac errors down the road

DIGIT= [0-9]
LETTER= [a-zA-Z]

YYINITIAL

{LETTER}
({LETTER}|{DIGIT})*

Line Counting Example

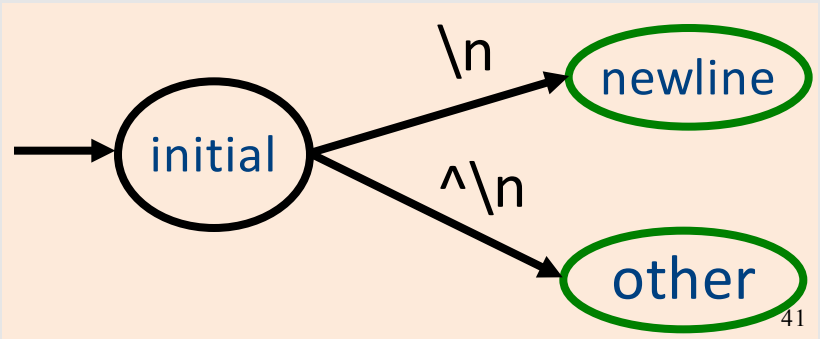
- Create a program that counts the number of lines in a given input text file

Creating a Scanner using JFlex

```
import java_cup.runtime.*;
%%
%cup
%{
  private int lineCounter = 0;
%}

%eofval{
  System.out.println("line number=" + lineCounter);
  return new Symbol(sym.EOF);
%eofval}

NEWLINE=\n
%%
{NEWLINE}       { lineCounter++; }
[^{NEWLINE}]    { }
```



Catching errors

- What if input doesn't match any token definition?
- Trick: Add a "catch-all" rule that matches any character and reports an error
  - Add after all other rules

A JFlex specification of C Scanner

```
import java_cup.runtime.*;
%%
%cup
%{
  private int lineCounter = 0;
%}
Letter= [a-zA-Z_]
Digit= [0-9]
%%
”\t”     { }
”\n”     { lineCounter++; }
“;”      { return new  Symbol(sym.SemiColumn);}
“++”     { return new  Symbol(sym.PlusPlus); }
“+=”     { return new  Symbol(sym.PlusEq); }
“+”      { return new  Symbol(sym.Plus); }
“while” { return new  Symbol(sym.While); }
{Letter}({Letter}|{Digit})*
         { return new  Symbol(sym.Id, yytext() ); }
“<=”     { return new  Symbol(sym.LessOrEqual); }
“<”      { return new  Symbol(sym.LessThan); }
```

Missing

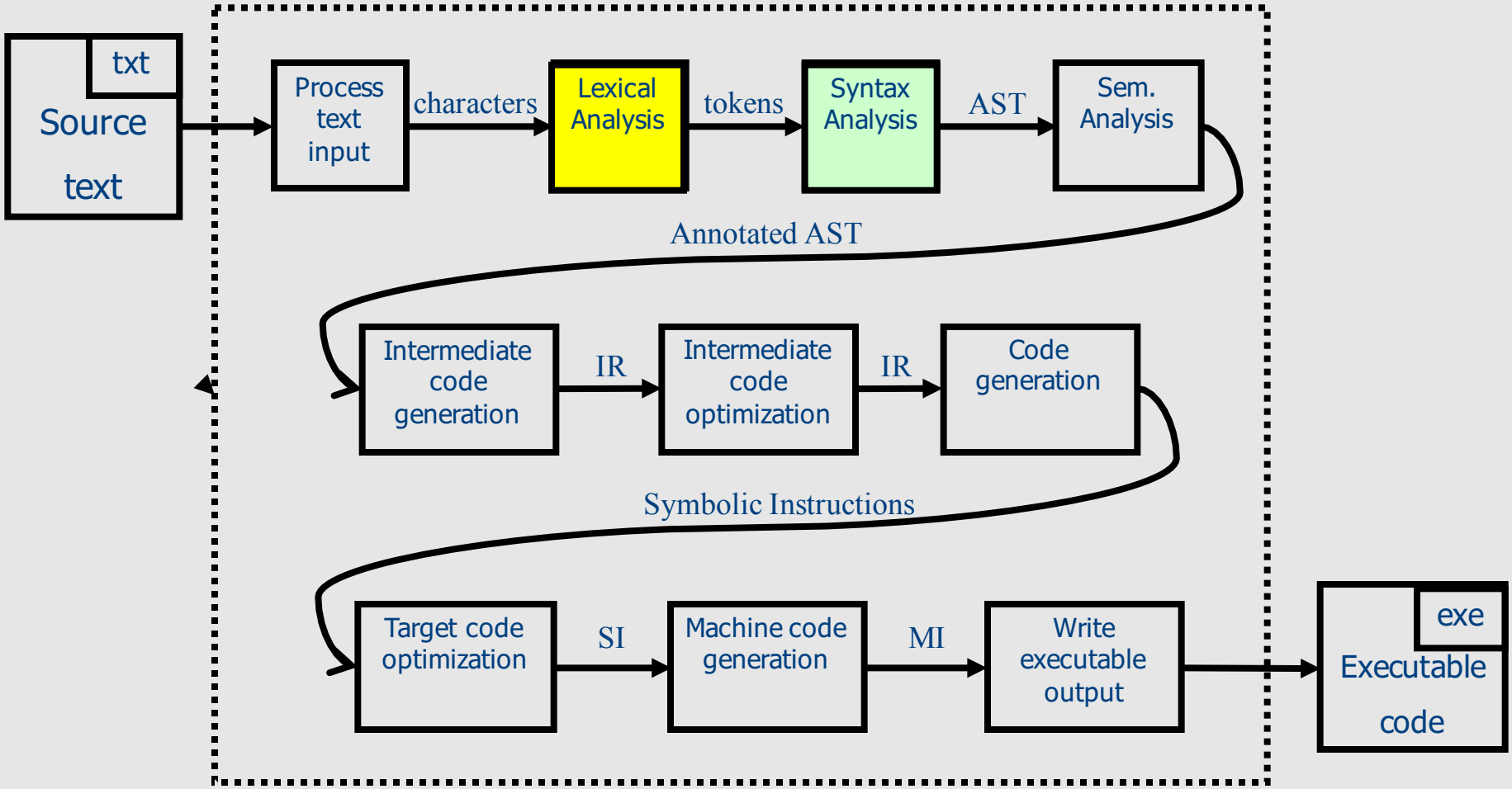
- Creating a lexical analysis by hand
- Table compression
- Symbol Tables
- Nested Comments
- Handling Macros

Lexical Analysis: Why

- Read input file
- Identify language keywords and standard identifiers
- Handle include files and macros
- Count line numbers
- Remove whitespaces
- Report illegal symbols

- [Produce symbol table]

The Real Anatomy of a Compiler



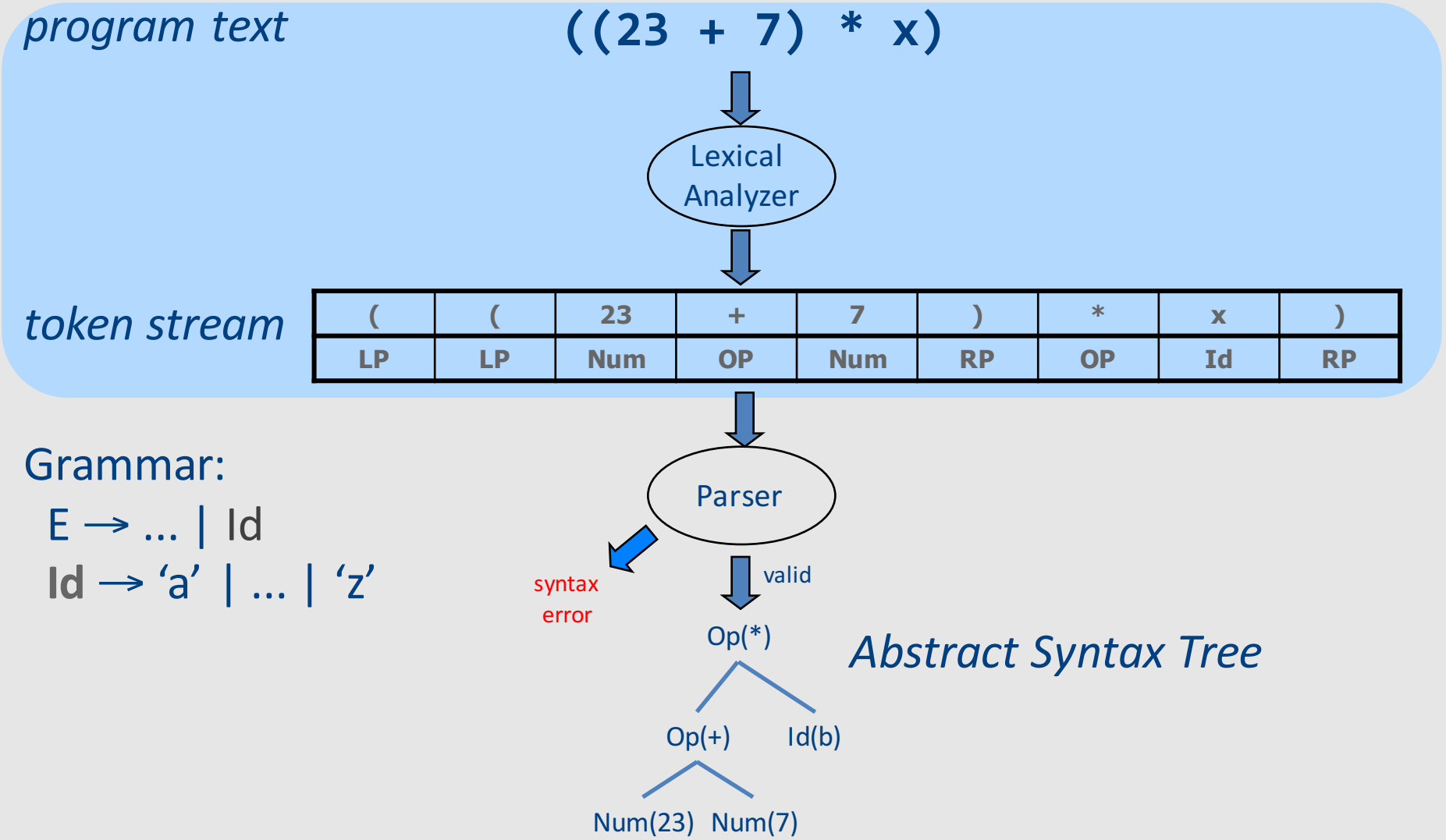
Syntax Analysis (1)

Context Free Languages

Context Free Grammars

Pushdown Automata

Frontend: Scanning & Parsing



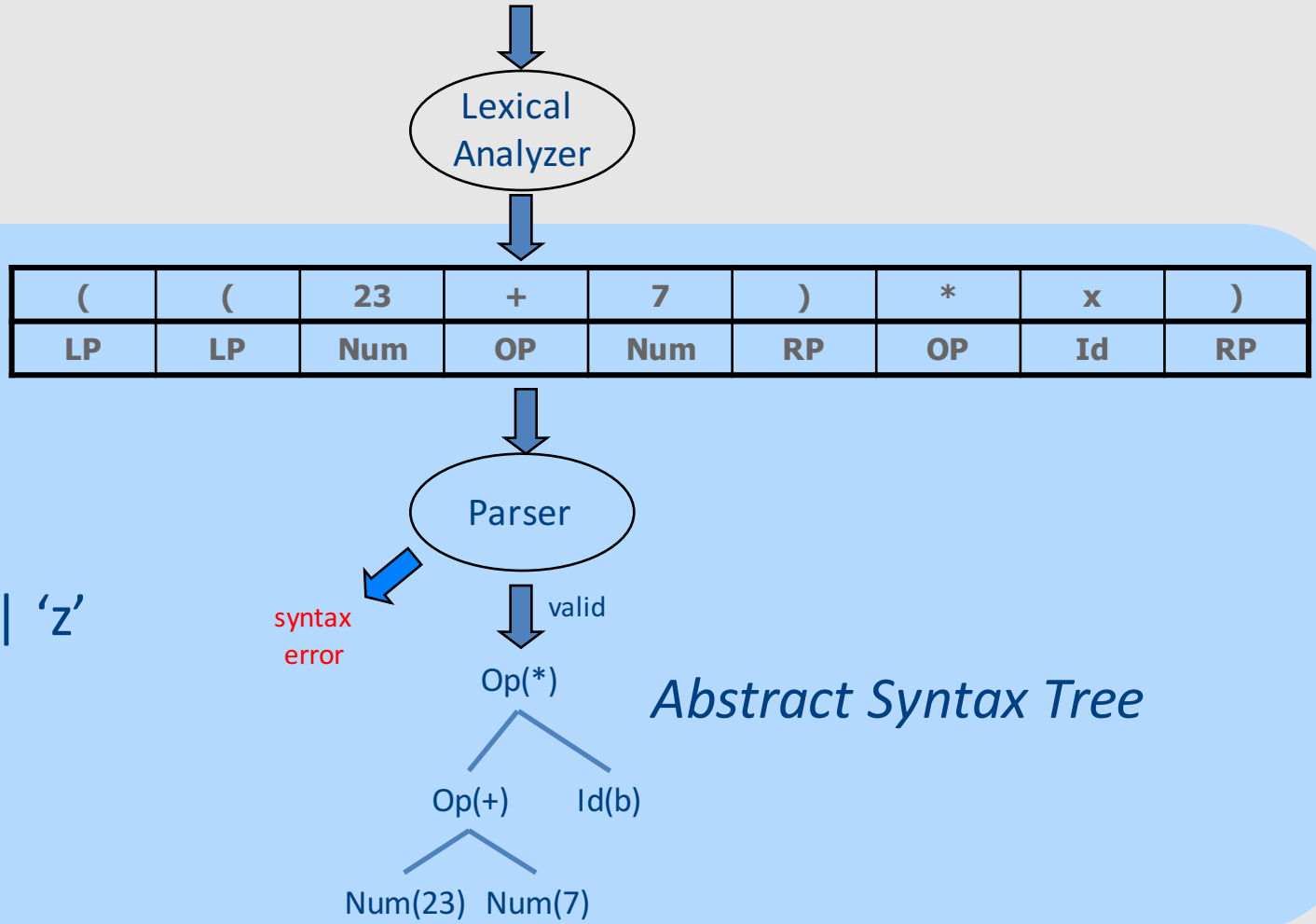
Grammar:

E → ... | Id

Id → ‘a’ | ... | ‘z’

From scanning to parsing

program text **((23 + 7) * x)**



Grammar:

E → ... | Id

Id → ‘a’ | ... | ‘z’

Abstract Syntax Tree

Parsing

- Construct a structured representation of the input

- Challenges
  - How do you describe the programming language?
  - How do you check validity of an input?
    - Is a sequence of tokens a valid program in the language?
  - How do you construct the structured representation?
  - Where do you report an error?

Some foundations

Context free languages (CFLs)

- $L_{01} = \{ 0^n 1^n \mid n > 0 \}$
- $L_{polyndrom} = \{pp' \mid p \in \Sigma^* , p'=reverse(p)\}$
- $L_{polyndrom\#} = \{p\#p' \mid p \in \Sigma^* , p'=reverse(p), \# \notin \Sigma\}$

Context free grammars (CFG)

G = (V,T,P,S)

- **V** – non terminals (syntactic variables)
- **T** – terminals (tokens)
- **P** – derivation rules
  - Each rule of the form V → (T ∪ V)*
- **S** – start symbol

What can CFGs do?

- Recognize CFLs

- S $\rightarrow$ 0T1
- T $\rightarrow$ 0T1 | ε

Recognizing CFLs

- Context Free Grammars (CFG)

$\sim$ language-defining power

- Nondeterministic push down automata (PDA)

$\not\sim$

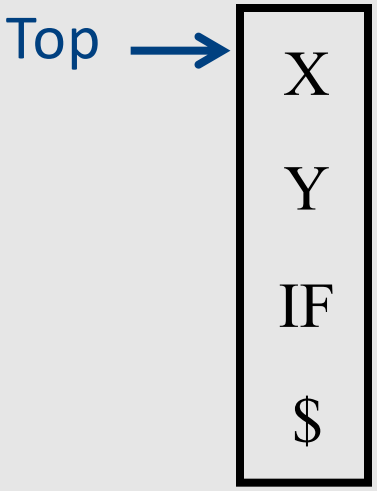
- Deterministic push down automata

Pushdown Automata (PDA)

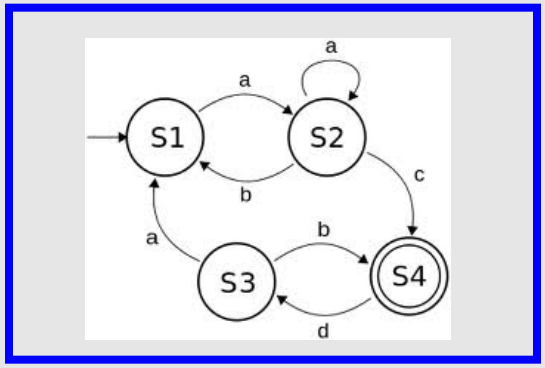
- Nondeterministic PDAs define all CFLs
- Deterministic PDAs model parsers.
  - Most programming languages have a deterministic PDA
  - Efficient implementation

Intuition: PDA

- An ε-NFA with the additional power to manipulate **one** stack



stack



control (ε-NFA)

Intuition: PDA

- Think of an ε-NFA with the additional power that it can manipulate a stack
- PDA moves are determined by:
  - The current state (of its "ε-NFA")
  - The current input symbol (or ε)
  - The current symbol on top of its stack

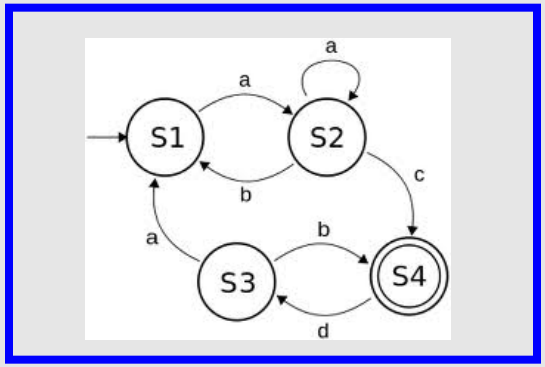
Intuition: PDA

Current

input `if (oops) then stat:= blah else abort`

Top → X Y IF $

stack



control (ε-NFA)

Intuition: PDA

- Moves:
  - Change state
  - Replace the top symbol by 0…n symbols
    - 0 symbols = "pop" ("reduce")
    - 0 < symbols = sequence of "pushes" ("shift")
- Nondeterministic choice of next move

PDA Formalism

- PDA = (Q, Σ, Γ, δ, q_0, $, F):
  - Q: finite set of states
  - Σ: Input symbols alphabet
  - Γ: stack symbols alphabet
  - δ: transition function
  - q_0: start state
  - $: start symbol
  - F: set of final states

Tokens

Non terminals

The Transition Function

- $\delta(q, a, X) = \{ (p_1, \sigma_1), \ldots, (p_n, \sigma_n)\}$
  - Input: triplet
    - A state $q \in Q$
    - An input symbol $a \in \Sigma$ or ε
    - A stack symbol $X \in \Gamma$
  - Output: set of 0 … k actions of the form (p, σ)
    - A state $p \in Q$
    - σ a sequence $X_1 \cdots X_n \in \Gamma^*$ of stack symbols

Actions of the PDA

- Say $(p, \sigma) \in \delta(q, a, X)$
  - If the PDA is in state q and X is the top symbol and a is at the front of the input
  - Then it can
    - Change the state to p.
    - Remove a from the front of the input
      - (but a may be ε).
    - Replace X on the top of the stack by σ.

Example: Deterministic PDA

- Design a PDA to accept $\{0^n1^n \mid n > 1\}$.
- The states:
  - q = We have not seen 1 so far
    - start state
  - p = we have seen at least one 1 and no 0s since
  - f = final state; accept.

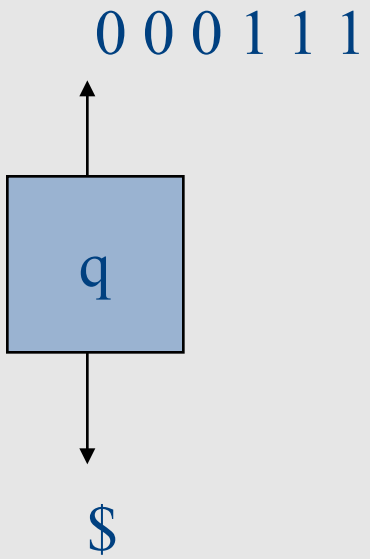
Example: Stack Symbols

- $ = start symbol.
  - Also marks the bottom of the stack,
  - Indicates when we have counted the same number of 1's as 0's.
- X = "counter"
  - used to count the number of 0s we saw

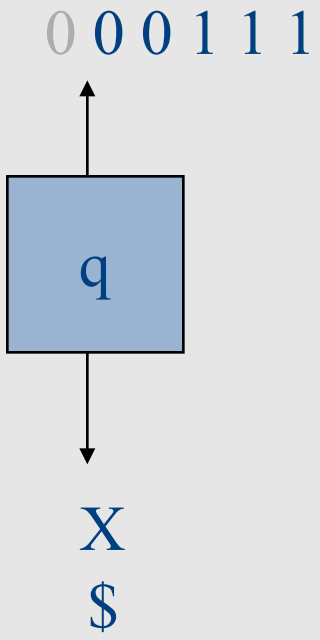
Example: Transitions

- $\delta(q, 0, \$) = \{(q, X\$)\}$.
- $\delta(q, 0, X) = \{(q, XX)\}$.
  - These two rules cause one X to be pushed onto the stack for each 0 read from the input.
- $\delta(q, 1, X) = \{(p, \varepsilon)\}$.
  - When we see a 1, go to state p and pop one X.
- $\delta(p, 1, X) = \{(p, \varepsilon)\}$.
  - Pop one X per 1.
- $\delta(p, \varepsilon, \$) = \{(f, \$)\}$.
  - Accept at bottom.

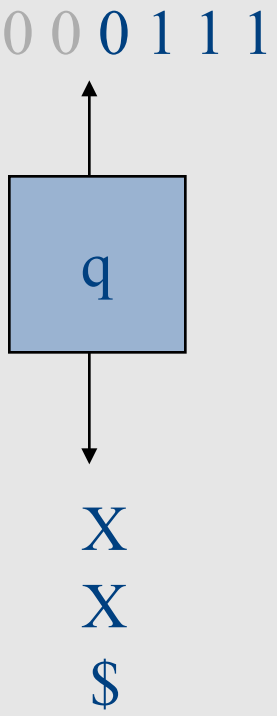
Actions of the Example PDA



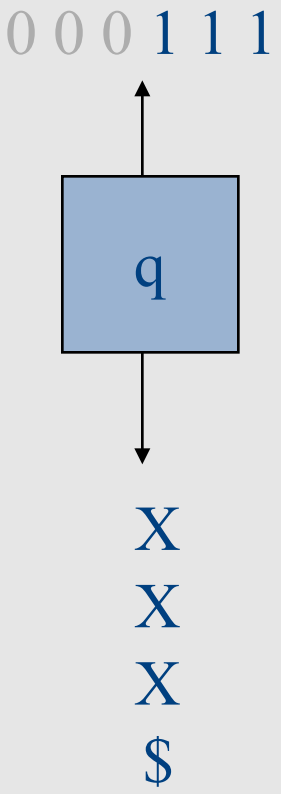
Actions of the Example PDA



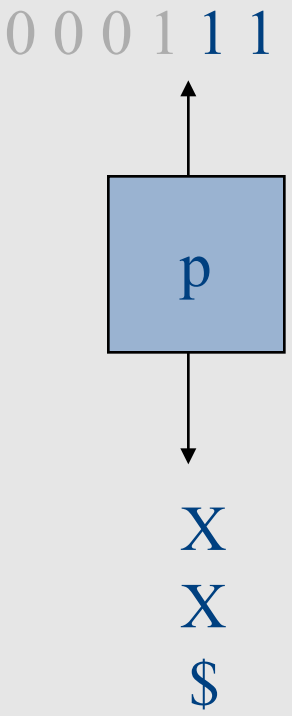
Actions of the Example PDA



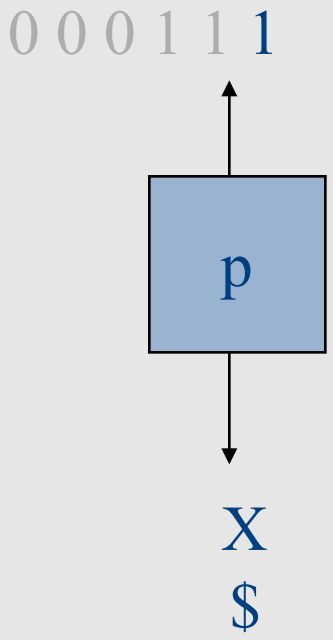
Actions of the Example PDA



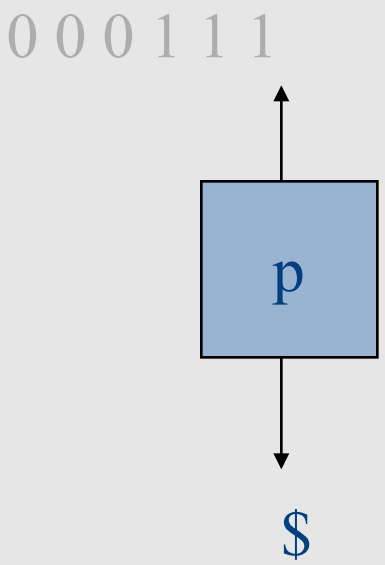
Actions of the Example PDA



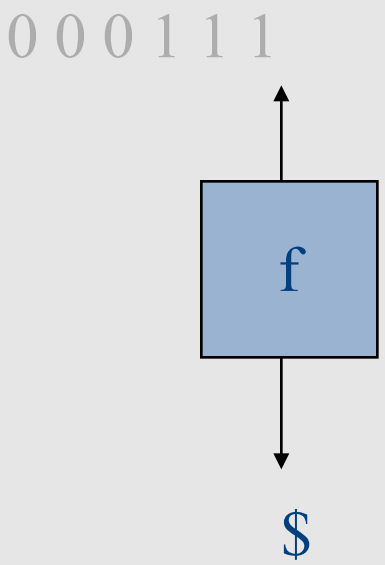
Actions of the Example PDA



Actions of the Example PDA



Actions of the Example PDA



Example: Non Deterministic PDA

- A PDA that accepts palindromes
  - L {pp' $\in \Sigma^*$ | p'=reverse(p)}

Here we are in 0368-3133

Broad kinds of parsers

- Parsers for arbitrary grammars
  - Earley's method, CYK method
  - Usually, not used in practice (though might change)
- Top-Down parsers
  - Construct parse tree in a top-down matter
  - Find the leftmost derivation
- Bottom-Up parsers
  - Construct parse tree in a bottom-up manner
  - Find the rightmost derivation in a reverse order

The plan

- Context free grammars
- Top down parsing: Intuition
- Brute force parsing
- Predictive parsing
- Earley parsing

Context free grammars (CFGs)

Context free grammars (CFGs)

$$G = (V,T,P,S)$$

- **V** – non terminals (syntactic variables)
- **T** – terminals (tokens)
- **P** – derivation rules
  - Each rule of the form $V \rightarrow (T \cup V)^*$
- **S** – start symbol

Example grammar

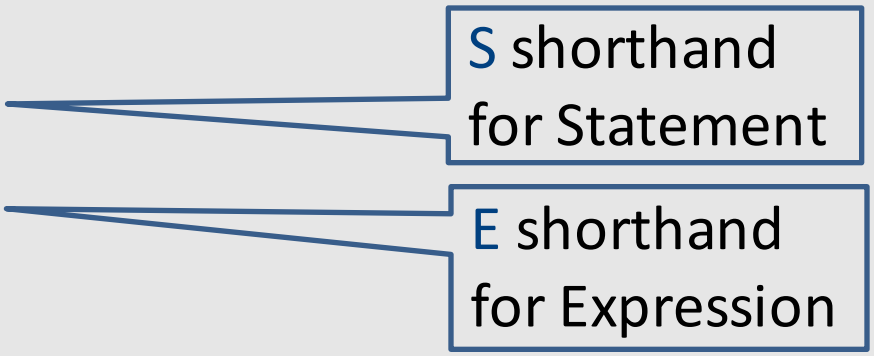
$S \rightarrow S\ ;\ S$

$S \rightarrow id := E$

$E \rightarrow id$

$E \rightarrow num$

$E \rightarrow E + E$



CFG terminology

S → S **;** S
S → id **:=** E
E → **id**
E → **num**
E → E **+** E

Symbols:
Terminals (tokens): **; := ( ) id num print**
Non-terminals: S E L

Start non-terminal: S
Convention: the non-terminal appearing in the first derivation rule

Grammar productions (rules)
N → μ

CFG terminology

- Derivation - a sequence of replacements of non-terminals using the derivation rules
- Language - the set of strings of terminals derivable from the start symbol
- Sentential form - the result of a partial derivation
  - May contain non-terminals

Derivations

- Show that a sentence ω is in a grammar G
  - Start with the start symbol
  - Repeatedly replace one of the non-terminals by a right-hand side of a production
  - Stop when the sentence contains only terminals
- Given a sentence αNβ and rule N→μ
  αNβ => αμβ
- ω is in L(G) if S =>* ω

Derivation

sentence

```
x := z;
y := x + z
```

grammar

```
S → S;S
S → id := E | …
E → id | E + E | E * E |…
```

```
          S
       S ; S
id := E ; S
id := E ; id := E
id := id ;id := E
id := id ;id := E + E
id := id ;id := E + id
id := id ;id := id + id
```

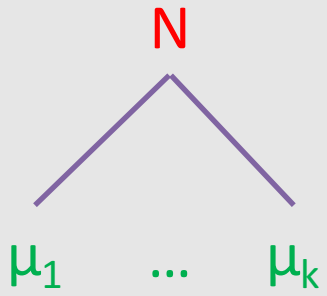
```
S → S;S
S → id := E
S → id := e
E → id
E → E + E
E → id
E → id
```

<id,"x"><ASS><id,"z"><SEMI><id,"y"><ASS><id,"x"><PLUS><id,"z">

Parse trees: Traces of Derivations

- Tree nodes are symbols, children ordered left-to-right
- Each internal node is non-terminal
  - Children correspond to one of its productions

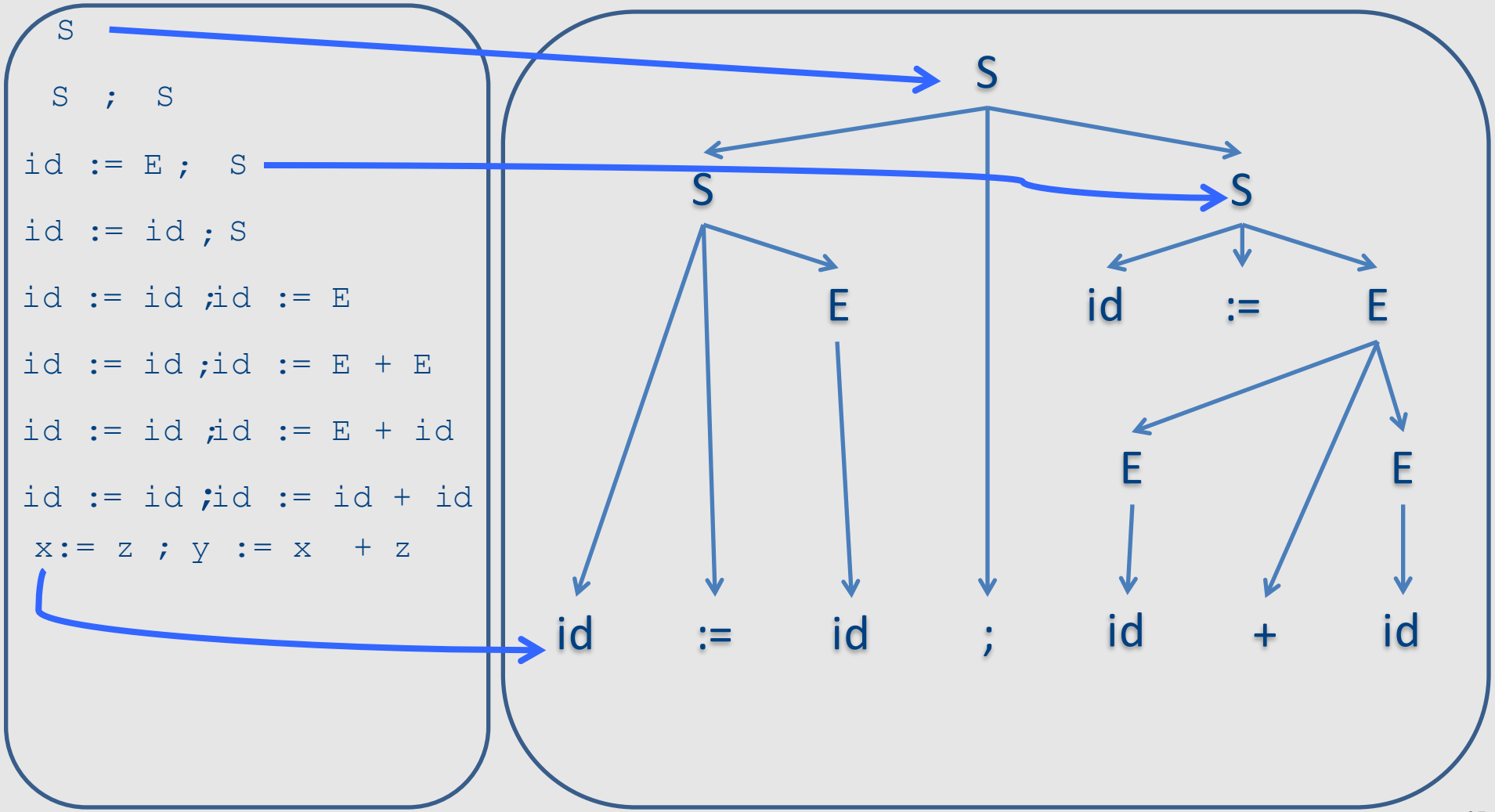
$N \rightarrow \mu_1 \ldots \mu_k$



- Root is start non-terminal
- Leaves are tokens

- ***Yield*** of parse tree: left-to-right walk over leaves

Parse Tree


S
S ; S
id := E ; S
id := id ; S
id := id ;id := E
id := id ;id := E + E
id := id ;id := E + id
id := id ;id := id + id
x:= z ; y := x + z
S
S
S
E
id
:=
E
E
E
id
:=
id
;
id
+
id

Language of a CFG

- A sentence ω is in L(G) (valid program) if
  - There exists a corresponding derivation
  - There exists a corresponding parse tree

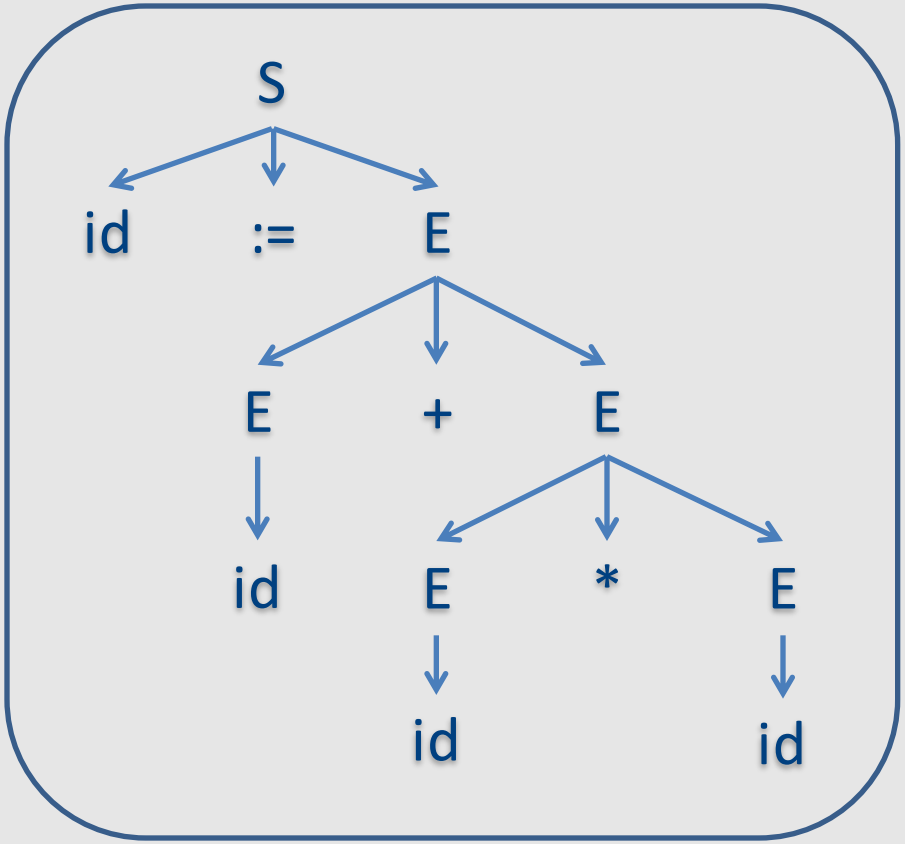
Questions

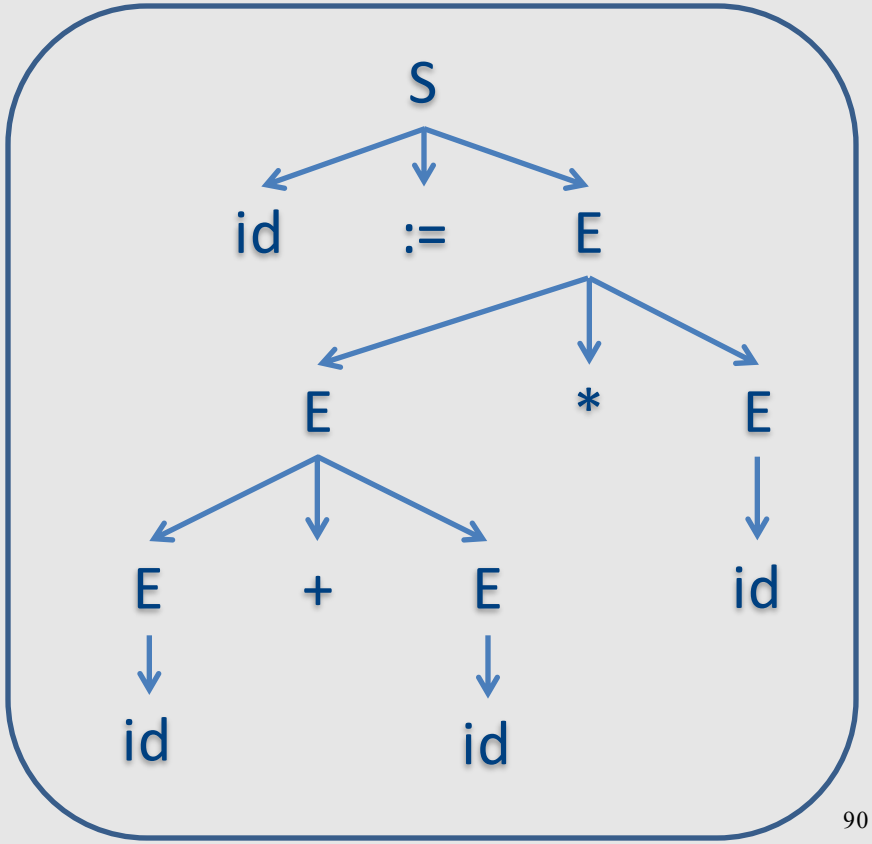
- How did we know which rule to apply on every step?
- Would we always get the same result?
- Does it matter?

Ambiguity

x := y+z*w

```
S → S ; S
S → id := E | …
E → id | E + E | E * E | …
```





Leftmost/rightmost Derivation

- Leftmost derivation
  - always expand leftmost non-terminal
- Rightmost derivation
  - Always expand rightmost non-terminal

- Allows us to describe derivation by listing the sequence of rules
  - always know what a rule is applied to

Leftmost Derivation

```
x := z;
y := x + z
```

```
S → S;S
S → id := E
E → id | E + E | E * E | ( E )
```

```
          S
        S ; S
  id := E ; S
  id := id ; S
  id := id ; id := E
  id := id; id := E + E
  id := id ; id := id + E
  id := id; id := id + id
    x := z   ;  y  :=  x + z
```

```
S → S;S
S → id := E
E → id
S → id := E
E → E + E
E → id
E → id
```

Rightmost Derivation

<id,”x”> ASS <id,”z”> ;
<id,”y”> ASS <id,”x”> PLUS
<id,”z”>

S → S;S
S → id := E | …
E → id | E + E | E * E | …

S
S ; S
S ; id := E
S ; id := E + E
S ; id := E + id
S ; id := id + id
id := E ; id := id + id
id := id ; id := id + id

S → S;S
S → id := E
E → E + E
E → id
E → id
S → id := E
E → id

<id,”x”> ASS <id,”z”> ; <id,”y”> ASS <id,”x”> PLUS <id,”z”>

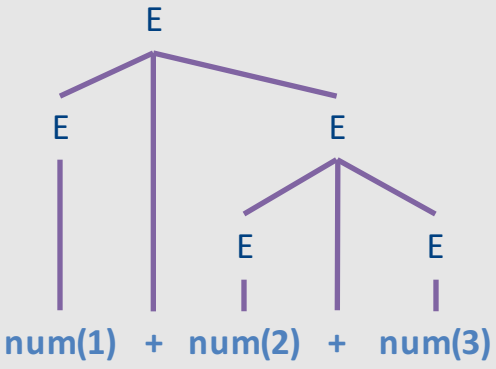
Sometimes there are two parse trees

Arithmetic expressions:

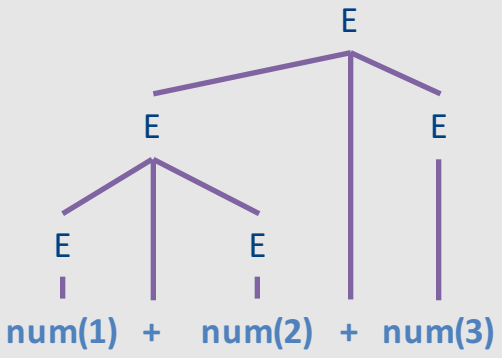
E → id
E → num
E → E + E
E → E * E
E → (E)

1 + 2 + 3

"1 + (2 + 3)"



"(1 + 2) + 3"



Leftmost derivation

E
E + E
num + E
num + E + E
num + num + E
num + num + num

Rightmost derivation

E
E + E
E + num
E + E + num
E + num + num
num + num + num

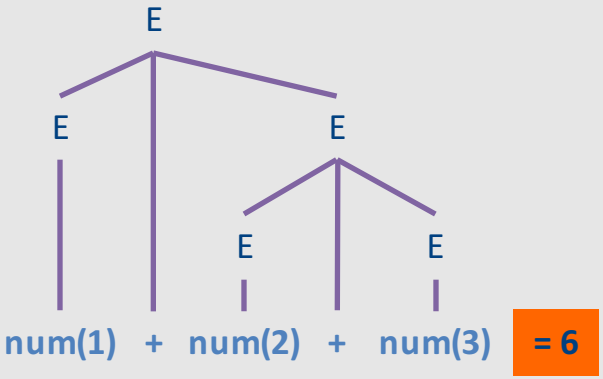
Is ambiguity a problem?

Arithmetic expressions:

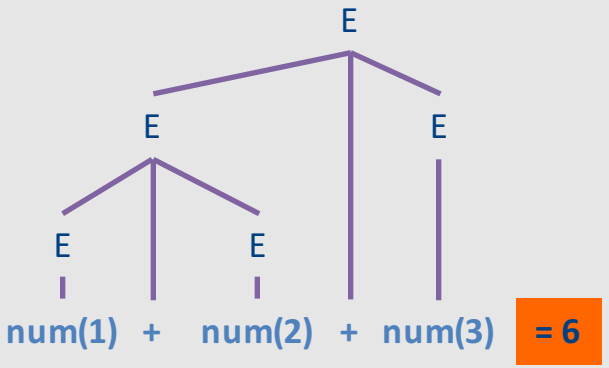
E → id
E → num
E → E + E
E → E * E
E → (E)

1 + 2 + 3

"1 + (2 + 3)"



"(1 + 2) + 3"



Leftmost derivation

E
E + E
num + E
num + E + E
num + num + E
num + num + num

Rightmost derivation

E
E + E
E + num
E + E + num
E + num + num
num + num + num

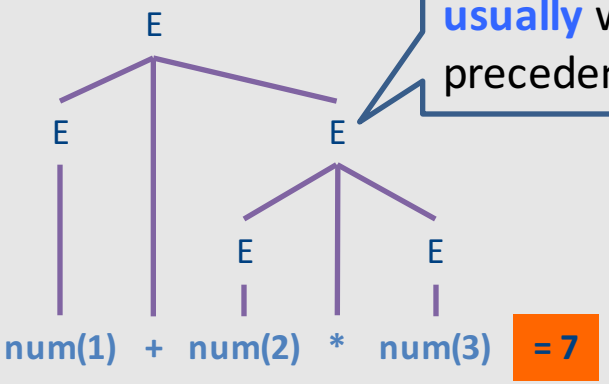
Problematic ambiguity example

Arithmetic expressions:

E → id
E → num
E → E + E
E → E * E
E → (E)

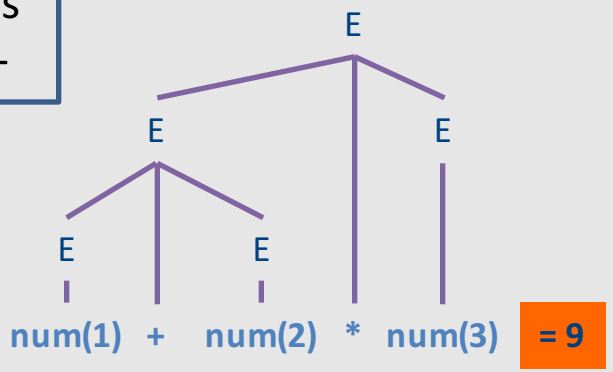
1 + 2 * 3

"1 + (2 * 3)"



This is what we **usually** want: * has precedence over +

"(1 + 2) * 3"



Leftmost derivation

E
E + E
num + E
num + E * E
num + num * E
num + num * num

Rightmost derivation

E
E * E
E * num
E + E * num
E + num * num
num + num * num

Ambiguous grammars

- A grammar is *ambiguous* if there exists a sentence for which there are
  - Two different leftmost derivations
  - Two different rightmost derivations
  - Two different parse trees
- Property of *grammars*, not *languages*

- Some languages are inherently ambiguous
  - No unambiguous grammars exist
- No algorithm to detect whether arbitrary grammar is ambiguous

Drawbacks of ambiguous grammars

- Ambiguous semantics
  - 1 + 2 * 3 = 7 or 9
- Parsing complexity
- May affect other phases
- Solutions
  - Transform grammar into non-ambiguous
  - Handle as part of parsing method
    - Using special form of “precedence”

Transforming ambiguous grammars to non-ambiguous by layering

Ambiguous grammar

E → E + E
E → E * E
E → id
E → num
E → (E)

Unambiguous grammar

E → E + T
E → T

T → T * F
T → F

F → id
F → num
F → (E)

Layer 1

Layer 2

Layer 3

Each layer takes care of one way of composing sub-strings to form a string:
1: by +
2: by *
3: atoms

Transformed grammar: * precedes +

Ambiguous grammar

E → E + E
E → E * E
E → id
E → num
E → (E)

Let's derive 1 + 2 * 3

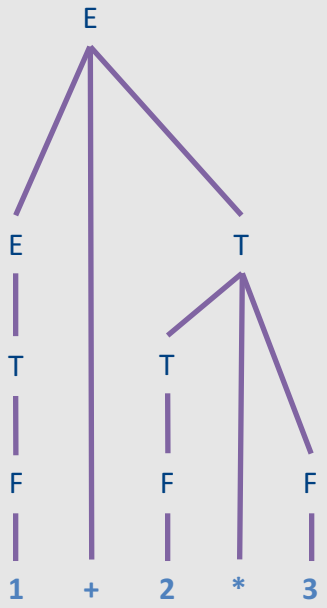
Unambiguous grammar

E → E + T
E → T
T → T * F
T → F
F → id
F → num
F → (E)

Derivation

E
=> E + T
=> T + T
=> F + T
=> 1 + T
=> 1 + T * F
=> 1 + F * F
=> 1 + 2 * F
=> 1 + 2 * 3

Parse tree



Transformed grammar: + precedes *

Ambiguous grammar

E → E + E
E → E * E
E → id
E → num
E → (E)

Let's derive 1 + 2 * 3

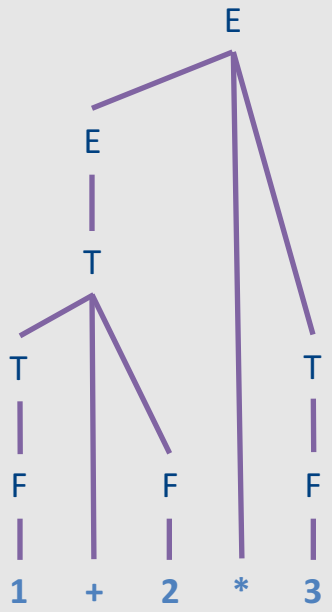
Unambiguous grammar

E → E * T
E → T
T → T + F
T → F
F → id
F → num
F → (E)

Derivation

E
=> E * T
=> T * T
=> T + F * T
=> F + F * T
=> 1 + F * T
=> 1 + 2 * T
=> 1 + 2 * F
=> 1 + 2 * 3

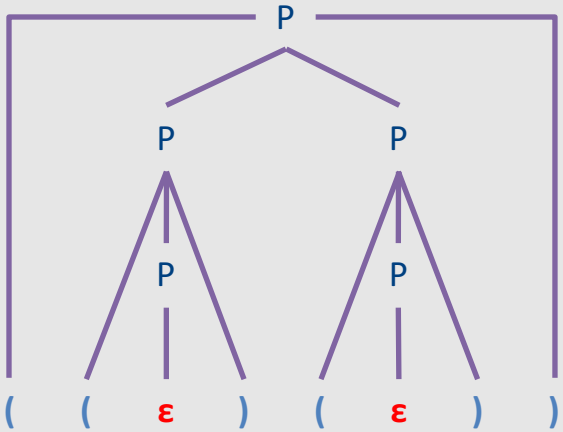
Parse tree

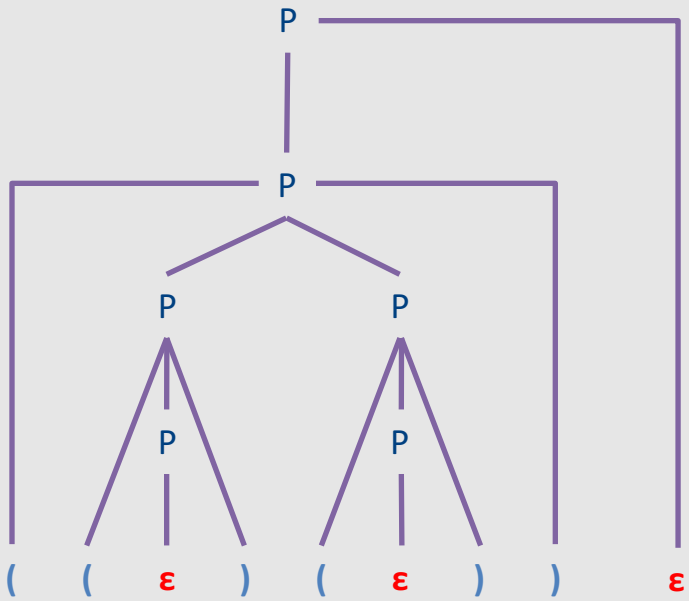


Another example for layering

Ambiguous grammar

P ⟶ ε
| P P
| (P)





Another example for layering

Ambiguous grammar

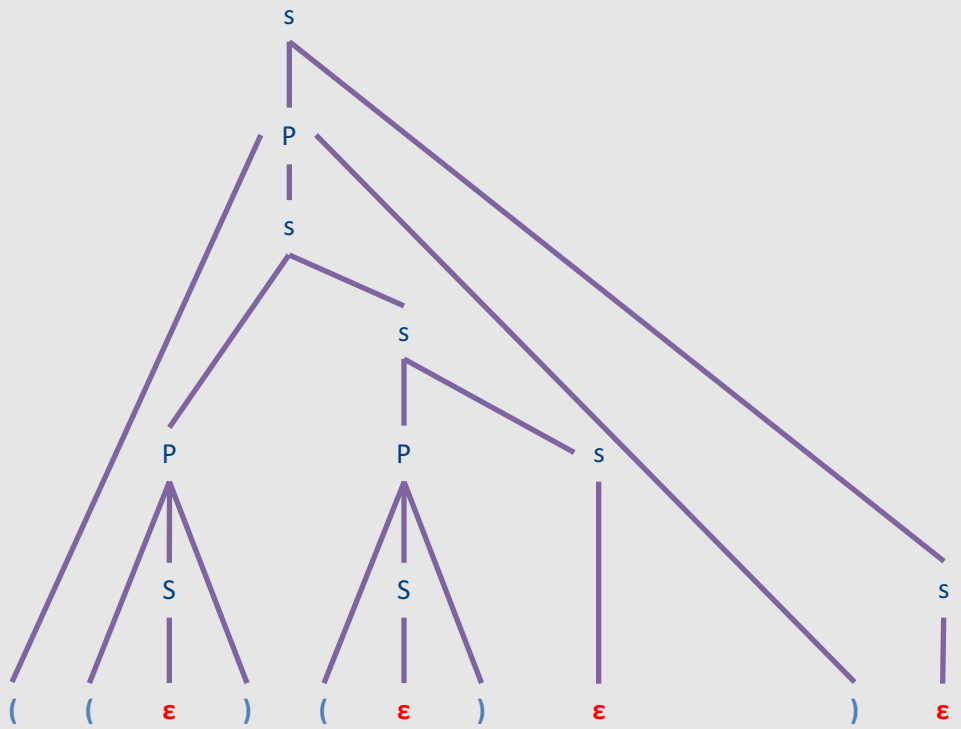
P → ε
| P P
| (P)

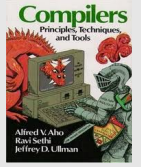
Unambiguous grammar

S → P S
| ε
P → (S)

Takes care of "concatenation"

Takes care of nesting





p. 174

"dangling-else" example

Ambiguous grammar

S → if E then S
S | if E then S else S
| other

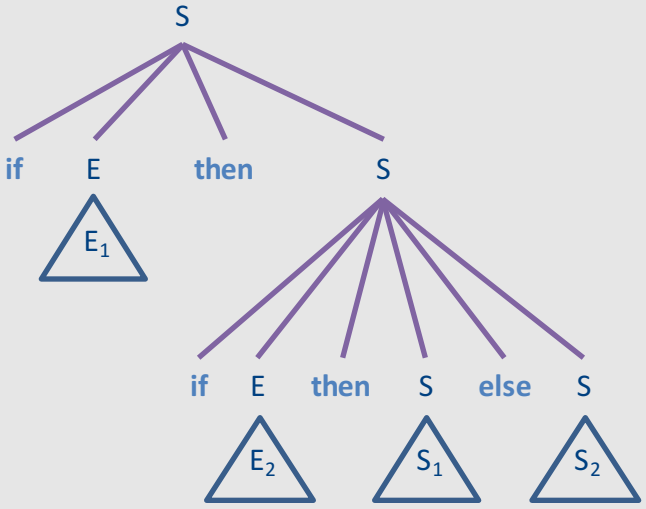
This is what we usually want: match **else** to closest unmatched **then**

Unambiguous grammar

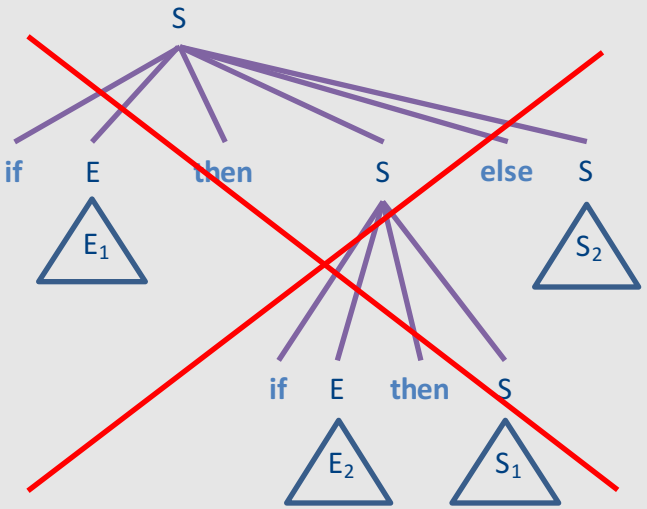
?

if E_1 then if E_2 then S_1 else S_2

if E_1 then (if E_2 then S_1 else S_2)



if E_1 then (if E_2 then S_1) else S_2



Broad kinds of parsers

- Parsers for arbitrary grammars
  - Earley's method, CYK method
  - Usually, not used in practice (though might change)
- Top-down parsers
  - Construct parse tree in a top-down matter
  - Find the leftmost derivation
- Bottom-up parsers
  - Construct parse tree in a bottom-up manner
  - Find the rightmost derivation in a reverse order

Intuition: Top-down parsing

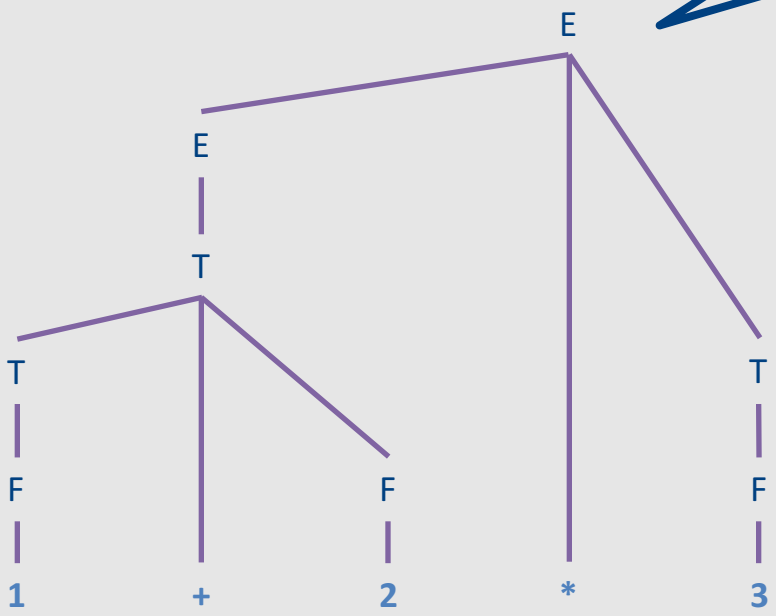
- Begin with start symbol
- Guess the productions
- Check if parse tree yields user's program

Intuition: Top-down parsing

Unambiguous grammar

E → E * T
E → T
T → T + F
T → F
F → id
F → num
F → (E)

Recall: Non standard precedence …



Intuition: Top-down parsing

Unambiguous grammar

E → E * T

E → T

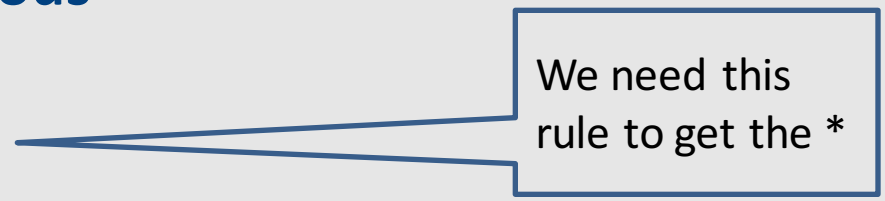
T → T + F

T → F

F → id

F → num

F → (E)



E

1 + 2 * 3

Intuition: Top-down parsing

Unambiguous grammar

E → E * T

E → T

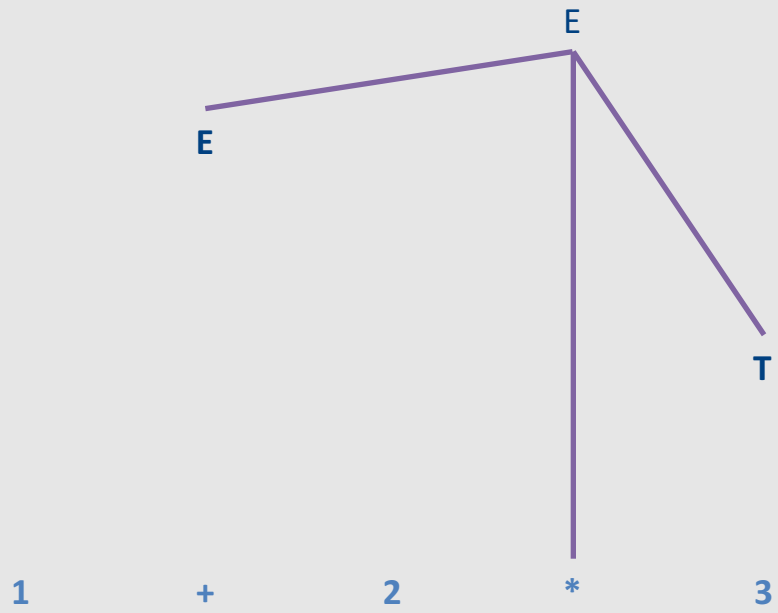
T → T + F

T → F

F → id

F → num

F → (E)



Intuition: Top-down parsing

Unambiguous grammar

E → E * T

E → T

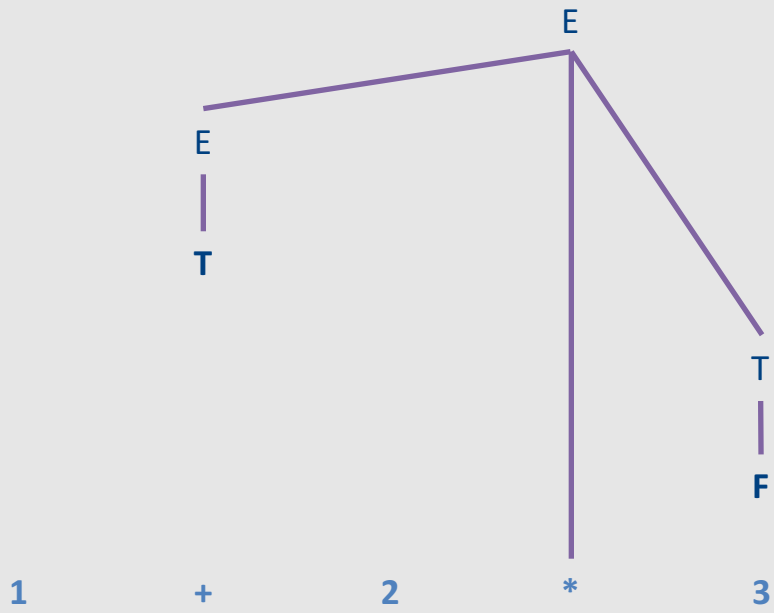
T → T + F

T → F

F → id

F → num

F → (E)



Intuition: Top-down parsing

Unambiguous grammar

E → E * T

E → T

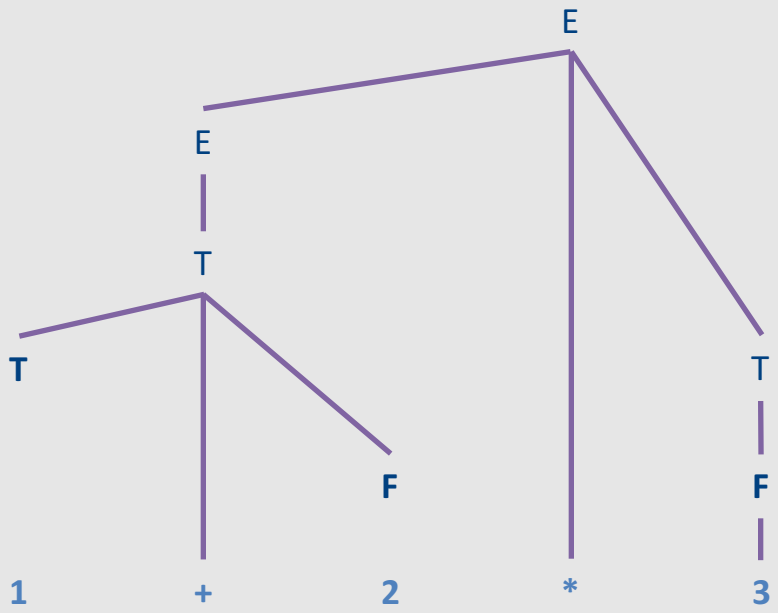
T → T + F

T → F

F → id

F → num

F → (E)



Intuition: Top-Down parsing

Unambiguous grammar

E → E * T

E → T

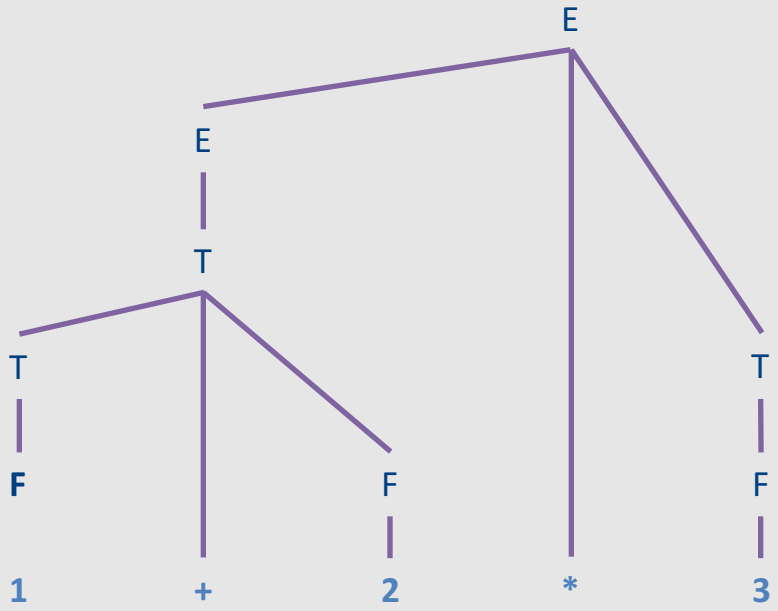
T → T + F

T → F

F → id

F → num

F → (E)

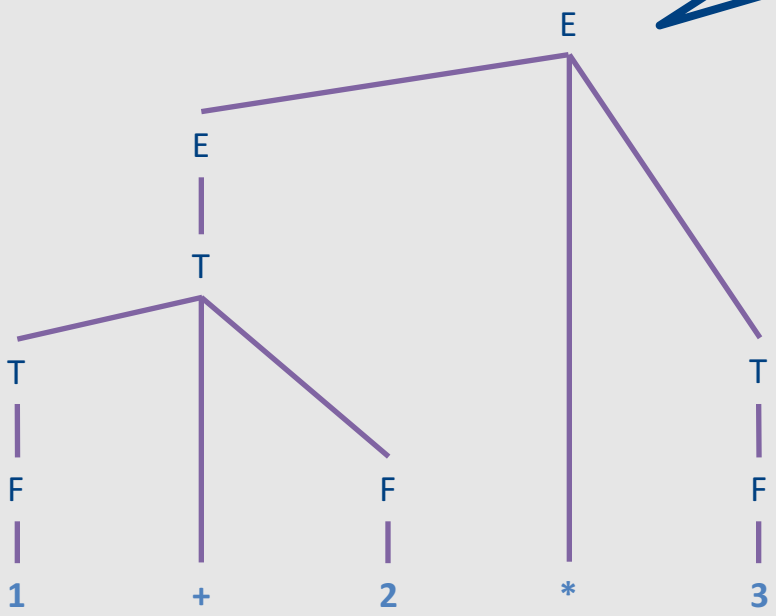


Intuition: Top-Down parsing

Unambiguous grammar

E → E * T
E → T
T → T + F
T → F
F → id
F → num
F → (E)

Recall: Non standard precedence …



Intuition: Bottom-up parsing

- Begin with the user's program
- Guess parse (sub)trees
- Check if root is the start symbol

Bottom-up parsing

Unambiguous grammar

E → E * T

E → T

T → T + F

T → F

F → id

F → num

F → (E)

1 + 2 * 3

Bottom-up parsing

Unambiguous grammar

E → E * T

E → T

T → T + F

T → F

F → id

F → num

F → (E)

Bottom-up parsing

Unambiguous grammar

E → E * T

E → T

T → T + F

T → F

F → id

F → num

F → (E)

Bottom-up parsing

Unambiguous grammar

E → E * T

E → T

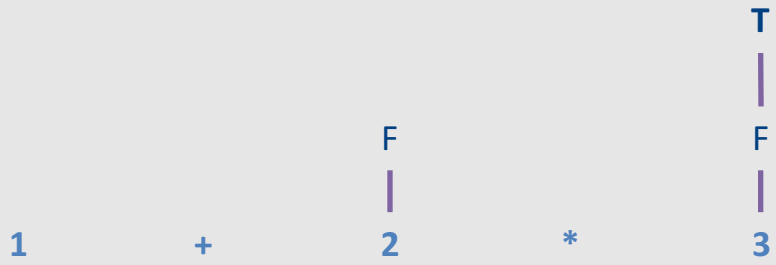
T → T + F

T → F

F → id

F → num

F → (E)



Bottom-up parsing

Unambiguous grammar

E → E * T

E → T

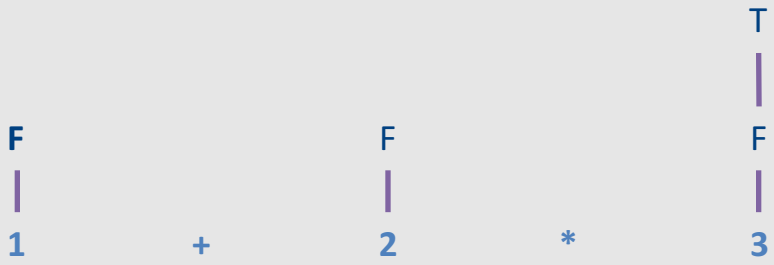
T → T + F

T → F

F → id

F → num

F → (E)



Bottom-up parsing

Unambiguous grammar

E → E * T

E → T

T → T + F

T → F

F → id

F → num

F → (E)

Bottom-up parsing

Unambiguous grammar

E → E * T

E → T

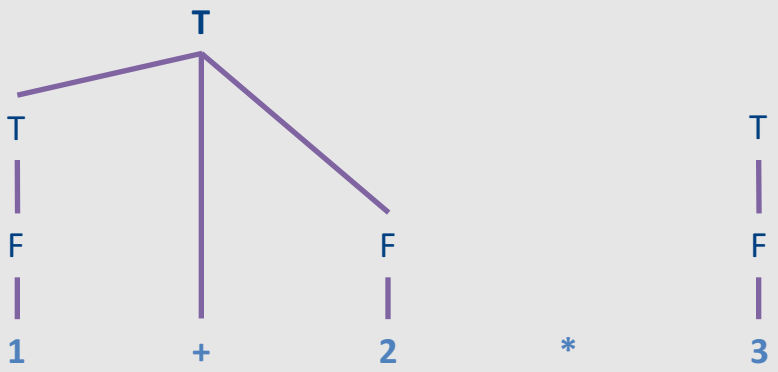
T → T + F

T → F

F → id

F → num

F → (E)



Bottom-up parsing

Unambiguous grammar

E → E * T

E → T

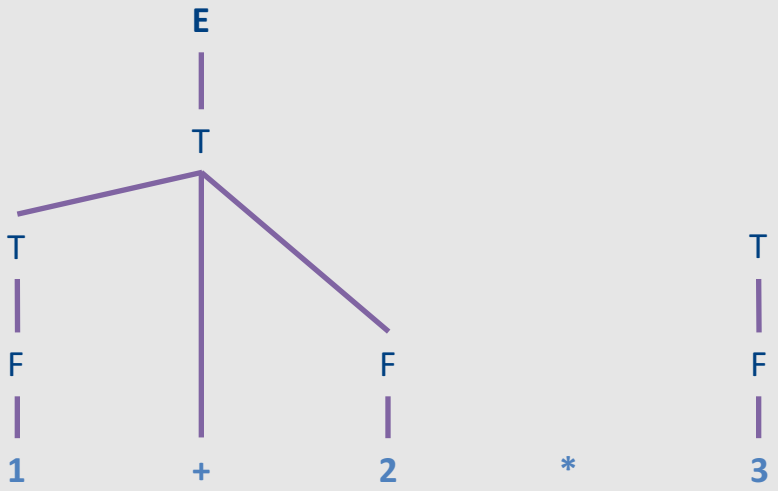
T → T + F

T → F

F → id

F → num

F → (E)

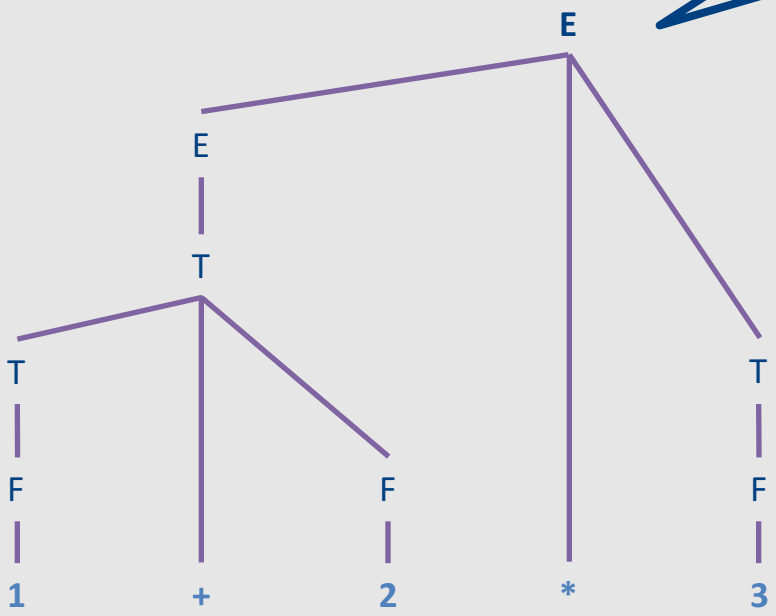


Bottom-up parsing

Unambiguous grammar

E → E * T
E → T
T → T + F
T → F
F → id
F → num
F → (E)

Recall: Non standard precedence …



Top-down parsing

Challenges in top-down parsing

- Top-down parsing begins with virtually no information
  - Begins with just the start symbol, which matches every program
- How can we know which productions to apply?

Which productions to apply?

- In general, we can‘t
  - There are some grammars for which the best we can do is guess and backtrack if we're wrong

“Brute-force” Parsing

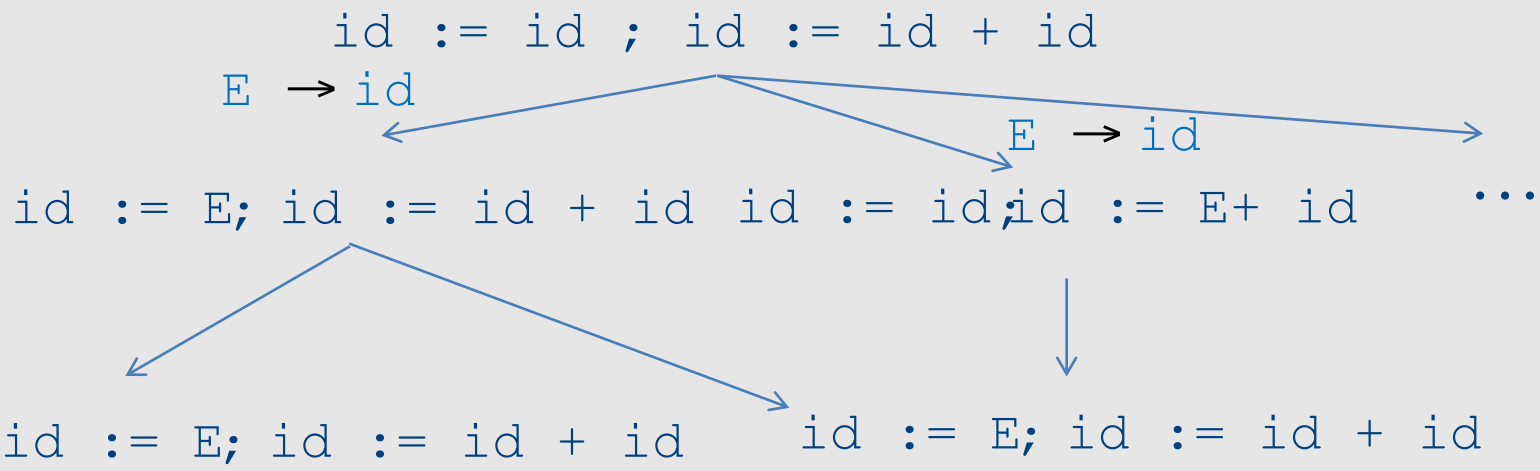
Which productions to apply?

- In general, we can‘t
  - There are some grammars for which the best we can do is guess and backtrack if we're wrong
- If we have to guess, how do we do it?
  - Parsing as a search algorithm
  - Too expensive in theory (exponential worst-case time) and practice

“Brute-force” Parsing

```
x := z;
y := x + z
```

```
S → S;S
S → id := E
E → id | E + E | …
```



(**not a parse tree**… a search for the parse tree by exhaustively applying all rules)

Predictive parsing

- Recursive descent
- LL(k) grammars

Predictive parsing

- Given a grammar G and a word w attempt to derive w using G
- Idea
  - Apply production to leftmost nonterminal
  - Pick production rule based on next input token
- General grammar
  - More than one option for choosing the next production based on a token
- Restricted grammars (LL)
  - Know exactly which single rule to apply
  - May require some lookahead to decide

Boolean expressions example

E → LIT | (E OP E) | **not** E
LIT → **true** | **false**
OP → **and** | **or** | **xor**

not (not true or false)

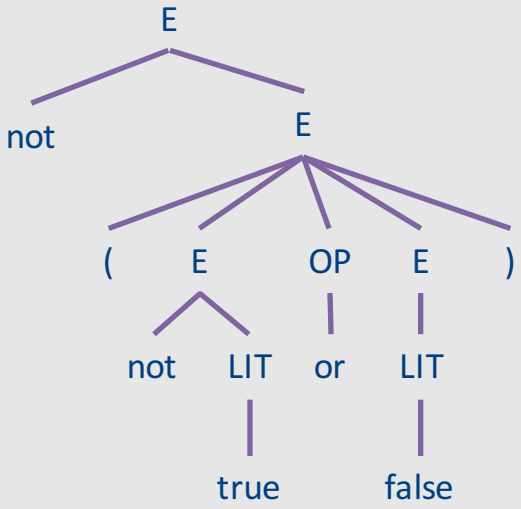
Boolean expressions example

E → LIT | (E OP E) | **not** E
LIT → **true** | **false**
OP → **and** | **or** | **xor**

production to apply known from next token

not (not true or false)

E =>
not E =>
not (E OP E) =>
not (not E OP E) =>
not (not LIT OP E) =>
not (not true OP E) =>
not (not true or E) =>
not (not true or LIT) =>
not (not true or false)



Recursive descent parsing

Recursive descent parsing

- Define a function for every nonterminal
- Every function work as follows
  - Find applicable production rule
  - Terminal function checks match with next input token
  - Nonterminal function calls (recursively) other functions
- If there are several applicable productions for a nonterminal, use lookahead

Matching tokens

E → LIT | (E OP E) | **not** E
LIT → **true** | **false**
OP → **and** | **or** | **xor**

```
match(token t) {
  if (current == t)
    current = next_token()
  else
    error
}
```

- Variable **`current`** holds the current input token

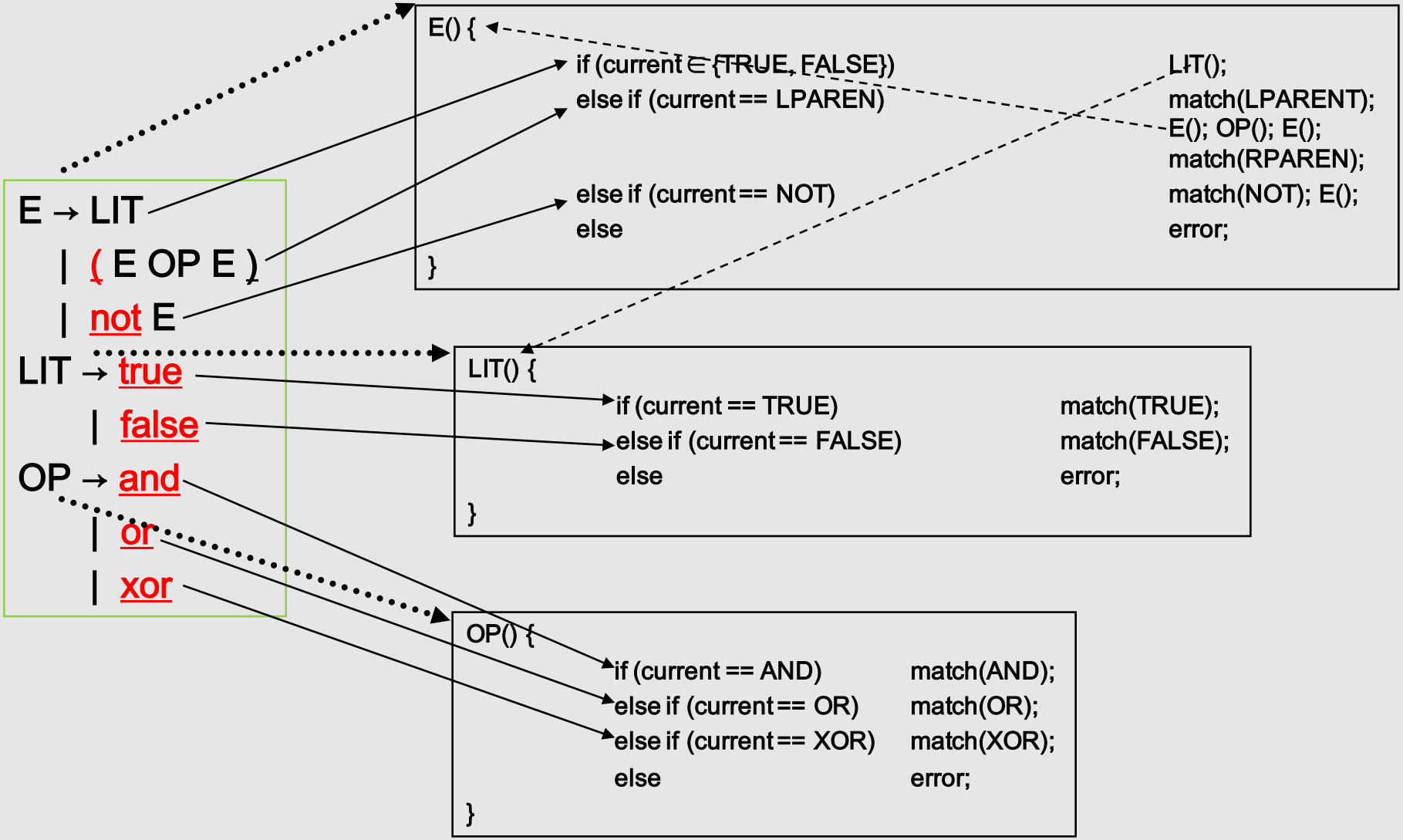
Functions for nonterminals

E → LIT | (E OP E) | **not** E
LIT → **true** | **false**
OP → **and** | **or** | **xor**

```
E() {
  if (current ∈ {TRUE, FALSE}) // E → LIT
    LIT();
  else if (current == LPAREN)  // E → ( E OP E )
   match(LPAREN); E(); OP(); E(); match(RPAREN);
  else if (current == NOT)     // E → not E
   match(NOT); E();
  else
   error;
}
```

```
LIT() {
  if (current == TRUE) match(TRUE);
  else if (current == FALSE) match(FALSE);
  else error;
}
```

Implementation via recursion



Recursive descent

```
void A() {
  choose an A-production, A → X1X2…Xk;
  for (i=1; i≤ k; i++) {
    if (Xi is a nonterminal)
      call procedure Xi();
    elseif (Xi == current)
      advance input;
    else
      report error;
  }
}
```

- How do you pick the right A-production?
- Generally – try them all and use backtracking
- In our case – use lookahead

Problem 1: productions with common prefix

term → ID | indexed_elem
indexed_elem → ID [expr]

- The function for indexed_elem will never be tried…
  - What happens for input of the form **`ID[expr]`**

Problem 2: null productions

S → A a b
A → a | ε

```
int S() {
  return  A() && match(token('a')) && match(token('b'));
}
int A()  {
 return  match(token('a')) || 1;
}
```

- What happens for input "ab"?
- What happens if you flip order of alternatives and try "aab"?

Problem 3: left recursion

$E \rightarrow E - term \mid term$

```
int E() {
  return E() && match(token('-')) && term();
}
```

- What happens with this procedure?
- Recursive descent parsers cannot handle left-recursive grammars

FIRST sets

X → YY | Z Z | Y Z | 1 Y

Y → 4 | ε

Z → 2

L(Z) = {2}

L(Y) = {4, ε}

L(Y) = {44, 4, ε, 22, 42, 2, 14, 1}

FIRST sets

X → YY | Z Z | Y Z | 1 Y

Y → 4 | ε

Z → 2

L(Z) = {2}

L(Y) = {4, ε}

L(Y) = {44, 4, ε, 22, 42, 2, 14, 1}

FIRST sets

- FIRST(X) = { t | X →* t β} ∪ {ℇ | X →* ℇ}
  - FIRST(X) = all terminals that α can appear as first in some derivation for X
    - + ℇ if can be derived from X

- Example:
  - FIRST(LIT) = { true, false }
  - FIRST((E OP E)) = { '(' }
  - FIRST(not E) = { not }

FIRST sets

- No intersection between FIRST sets => can always pick a single rule
- If the FIRST sets intersect, may need longer lookahead
  - LL(k) = class of grammars in which production rule can be determined using a lookahead of k tokens
  - LL(1) is an important and useful class

Computing FIRST sets

- FIRST (t) = { t } // "t" non terminal

- $\varepsilon \in$ FIRST(X) if
  - X → ε or
  - X → A_1 .. A_k and $\varepsilon \in$ FIRST(A_i) i=1…k

- FIRST(α) $\subseteq$ FIRST(X) if
  - X→ A_1 .. A_k α and $\varepsilon \in$ FIRST(A_i) i=1…k

Computing FIRST sets

- Assume no null productions A ⍰ ε
  1. Initially, for all nonterminals A, set
     FIRST(A) = { t | A → tω *for some* ω }
  2. Repeat the following until no changes occur:
     for each nonterminal A
       for each production A → Bω
         set FIRST(A) = FIRST(A) ∪ FIRST(B)
- This is known as fixed-point computation

FIRST sets computation example

STMT → if EXPR then STMT
| while EXPR do STMT
| EXPR ;
EXPR → TERM -> id
| zero? TERM
| not EXPR
| ++ id
| -- id
TERM → id
| constant

STMT	EXPR	TERM

1. Initialization

STMT → if EXPR then STMT
| while EXPR do STMT
| EXPR ;
EXPR → TERM -> id
| zero? TERM
| not EXPR
| ++ id
| -- id
TERM → id
| constant

STMT	EXPR	TERM
if while	zero? Not ++ --	id constant

2. Iterate 1

STMT → if EXPR then STMT
| while EXPR do STMT
| EXPR ;
EXPR → TERM -> id
| zero? TERM
| not EXPR
| ++ id
| -- id
TERM → id
| constant

STMT	EXPR	TERM
if while	zero? Not ++ --	id constant
zero? Not ++ --		

2. Iterate 2

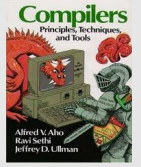
STMT → if EXPR then STMT
| while EXPR do STMT
| EXPR ;
EXPR → TERM -> id
| zero? TERM
| not EXPR
| ++ id
| -- id
TERM → id
| constant

STMT	EXPR	TERM
if while	zero? Not ++ --	id constant
zero? Not ++ --	id constant	

2. Iterate 3 – fixed-point

STMT → if EXPR then STMT
| while EXPR do STMT
| EXPR ;
EXPR → TERM -> id
| zero? TERM
| not EXPR
| ++ id
| -- id
TERM → id
| constant

STMT	EXPR	TERM
if while	zero? Not ++ --	id constant
zero? Not ++ --	id constant	
id constant		



p. 189

FOLLOW sets

- What do we do with nullable (ε) productions?
  - A → B C D B → ε C → ε
  - Use what comes afterwards to predict the right production
- For every production rule A → α
  - FOLLOW(A) = set of tokens that can immediately follow A
- Can predict the alternative A_k for a non-terminal N when the lookahead token is in the set
  - FIRST(A_k) → (if A_k is nullable then FOLLOW(N))

FOLLOW sets: Constraints

- $ ∈ FOLLOW(S)
- FIRST(β) – {ℰ} ⊆ FOLLOW(X)
  - For each A → α X β
- FOLLOW(A) ⊆ FOLLOW(X)
  - For each A → α X β and ℰ ∈ FIRST(β)

Example: FOLLOW sets

- E→ TX
- T→ (E) | int Y

X→+ E | ε

Y → * T | ε

Terminal	+	(	*	)	int
FOLLOW	int, (	int, (	int, (	_,), $	*,), +, $

Non. Term.	E	T	X	Y
FOLLOW	), $	+,), $	$,)	_,), $

Prediction Table

- $A \rightarrow \alpha$

- $T[A,t] = \alpha$ if $t \in FIRST(\alpha)$
- $T[A,t] = \alpha$ if $\varepsilon \in FIRST(\alpha)$ and $t \in FOLLOW(A)$
  - t can also be $

- T is not well defined → the grammar is not LL(1)

LL(k) grammars

- A grammar is in the class LL(K) when it can be derived via:
  - Top-down derivation
  - Scanning the input from left to right (L)
  - Producing the leftmost derivation (L)
  - With lookahead of k tokens (k)
- A language is said to be LL(k) when it has an LL(k) grammar

LL(1) grammars

- A grammar is in the class LL(K) iff
  - For every two productions $A \rightarrow \alpha$ and $A \rightarrow \beta$ we have
    - $FIRST(\alpha) \cap FIRST(\beta) = \{\}$ // including ε
    - If $\varepsilon \in FIRST(\alpha)$ then $FIRST(\beta) \cap FOLLOW(A) = \{\}$
    - If $\varepsilon \in FIRST(\beta)$ then $FIRST(\alpha) \cap FOLLOW(A) = \{\}$

Problem: Non LL Grammars

S → A a b
A → a | ε

```
bool S() {
  return  A() && match(token('a')) && match(token('b'));
}

bool A() {
 return  match(token('a')) || true;
}
```

- What happens for input "ab"?
- What happens if you flip order of alternatives and try "aab"?

Problem: Non LL Grammars

S → A a b
A → a | ε

- FIRST(S) = { a } FOLLOW(S) = { $ }
- FIRST(A) = { a, ε } FOLLOW(A) = { a }

- FIRST/FOLLOW conflict

Back to problem 1

term → ID | indexed_elem
indexed_elem → ID [expr]

- FIRST(term) = { ID }
- FIRST(indexed_elem) = { ID }

- FIRST/FIRST conflict

Solution: left factoring

- Rewrite the grammar to be in LL(1)

term → ID | indexed_elem
indexed_elem → ID [expr]

term → ID after_ID
After_ID → [expr] | ε

Intuition: just like factoring x*y + x*z into x*(y+z)

Left factoring – another example

S → if E then S else S
| if E then S
| T

S → if E then S S’
| T
S’ → else S | ε

Back to problem 2

S → A a b
A → a | ε

- FIRST(S) = { a } FOLLOW(S) = { }
- FIRST(A) = { a , ε } FOLLOW(A) = { a }

- FIRST/FOLLOW conflict

Solution: substitution

S → A a b
A → a | ε

Substitute A in S

S → a a b | a b

Left factoring

S → a after_A
after_A → a b | b

Back to problem 3

E → E - term | term

- Left recursion cannot be handled with a bounded lookahead
- What can we do?

Left recursion removal

$N \rightarrow N\alpha \mid \beta$

G_1

$N \rightarrow \beta N'$
$N' \rightarrow \alpha N' \mid \varepsilon$

G_2

Can be done algorithmically.
Problem: grammar becomes mangled beyond recognition

- $L(G_1) = \beta, \beta\alpha, \beta\alpha\alpha, \beta\alpha\alpha\alpha, \ldots$
- $L(G_2)$ = same

- For our 3rd example:

E → E - term | term

E → term TE | term
TE → - term TE | ε

LL(k) Parsers

- Recursive Descent
  - Manual construction
  - Uses recursion

- Wanted
  - A parser that can be generated automatically
  - Does not use recursion

LL(k) parsing via pushdown automata

- Pushdown automaton uses
  - Prediction stack
  - Input stream
  - Transition table
    - nonterminals x tokens -> production alternative
    - Entry indexed by nonterminal N and token t contains the alternative of N that must be predicated when current input starts with t

LL(k) parsing via pushdown automata

- Two possible moves
  - Prediction
    - When top of stack is nonterminal N, pop N, lookup table[N,t]. If table[N,t] is not empty, push table[N,t] on prediction stack, otherwise – syntax error
  - Match
    - When top of prediction stack is a terminal T, must be equal to next input token t. If (t == T), pop T and consume t. If (t ≠ T) syntax error
- Parsing terminates when prediction stack is empty
  - If input is empty at that point, success. Otherwise, syntax error

Example transition table

(1) E → LIT

(2) E → (E OP E)

(3) E → not E

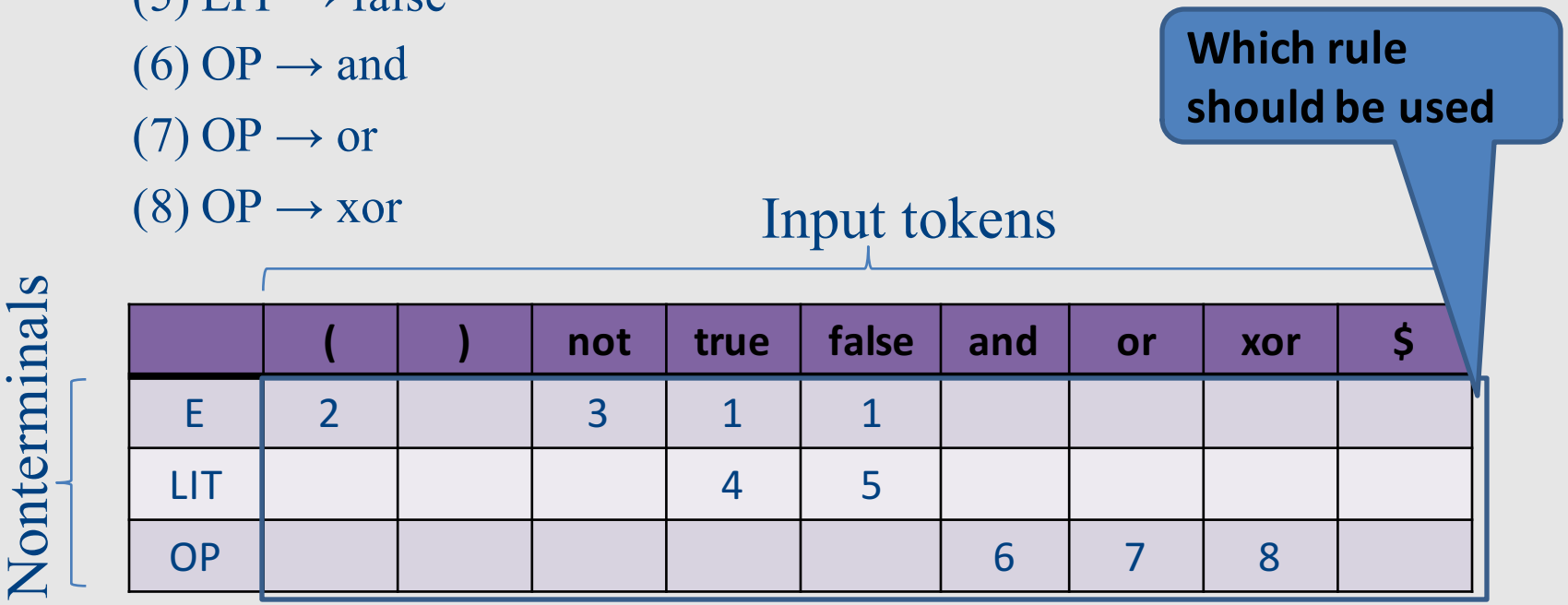
(4) LIT → true

(5) LIT → false

(6) OP → and

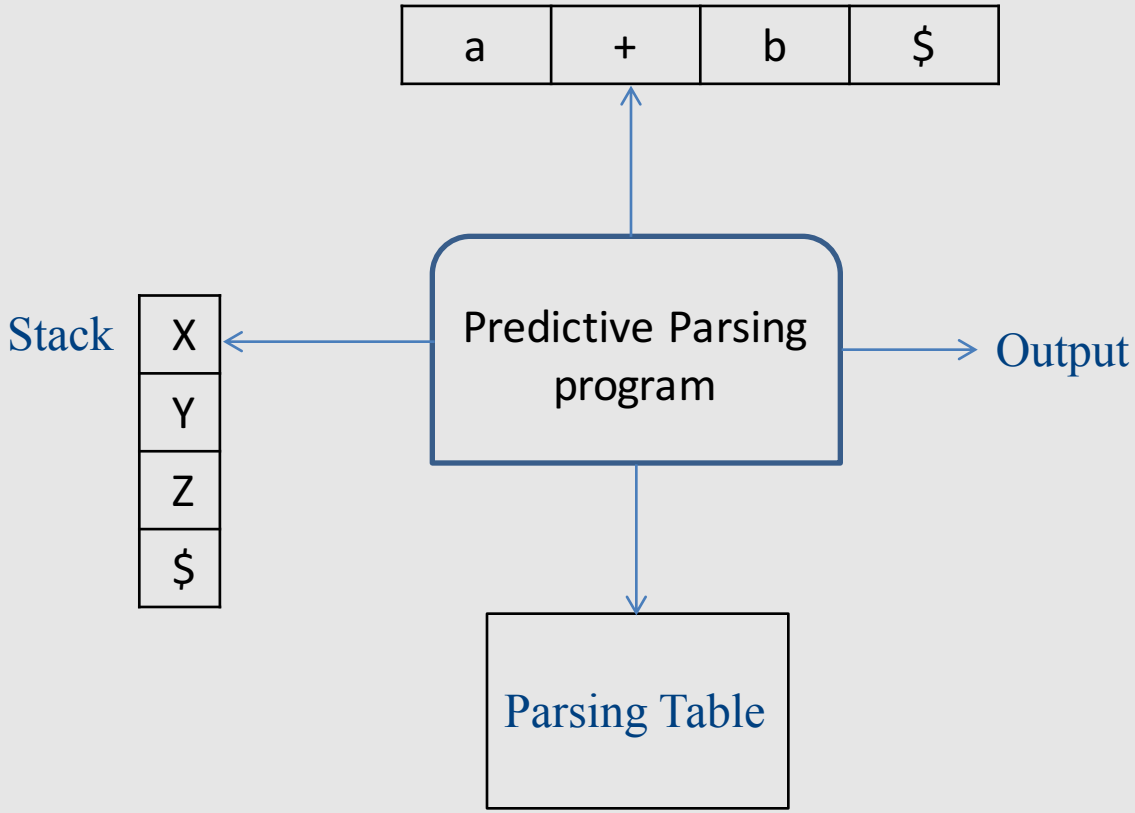
(7) OP → or

(8) OP → xor



	(	)	not	true	false	and	or	xor	$
E	2		3	1	1				
LIT				4	5				
OP						6	7	8	

Model of non-recursive predictive parser



Running parser example

aacbb$

A → aAb | c

Input suffix	Stack content	Move
aacbb$	A$	predict(A,a) = A → aAb
aacbb$	aAb$	match(a,a)
acbb$	Ab$	predict(A,a) = A → aAb
acbb$	aAbb$	match(a,a)
cbb$	Abb$	predict(A,c) = A → c
cbb$	cbb$	match(c,c)
bb$	bb$	match(b,b)
b$	b$	match(b,b)
$	$	match($,$) – success

	a	b	c
A	A → aAb		A → c

Erorrs

Handling Syntax Errors

- Report and locate the error
- Diagnose the error
- Correct the error
- Recover from the error in order to discover more errors
  - without reporting too many “strange” errors

Error Diagnosis

- Line number
  - may be far from the actual error
- The current token
- The expected tokens
- Parser configuration

Error Recovery

- Becomes less important in interactive environments
- Example heuristics:
  - Search for a semi-column and ignore the statement
  - Try to “replace” tokens for common errors
  - Refrain from reporting 3 subsequent errors
- Globally optimal solutions
  - For every input w, find a valid program w’ with a “minimal-distance” from w

Illegal input example

abcbb$

A → aAb | c

Input suffix	Stack content	Move
abcbb$	A$	predict(A,a) = A → aAb
abcbb$	aAb$	match(a,a)
bcbb$	Ab$	predict(A,b) = ERROR

	a	b	c
A	A → aAb		A → c

Error handling in LL parsers

c$

S → a c | b S

Input suffix	Stack content	Move
c$	S$	predict(S,c) = ERROR

- Now what?
  - Predict b S anyway "missing token b inserted in line XXX"

	a	b	c
S	S → a c	S → b S	

Error handling in LL parsers

c$

S → a c | b S

Input suffix	Stack content	Move
bc$	S$	predict(b,c) = S → bS
bc$	bS$	match(b,b)
c$	S$	Looks familiar?

- Result: infinite loop

	a	b	c
S	S → a c	S → b S	

Error handling and recovery

- x = a * (p+q * (-b * (r-s);
  - Where should we report the error?
  - The valid prefix property

The Valid Prefix Property

- For every prefix tokens
  - $t_1, t_2, \ldots, t_i$ that the parser identifies as legal:
    - there exists tokens $t_{i+1}, t_{i+2}, \ldots, t_n$ such that $t_1, t_2, \ldots, t_n$ is a syntactically valid program

- If every token is considered as single character:
  - For every prefix word u that the parser identifies as legal there exists w such that u.w is a valid program

The Valid Prefix Property

- For every prefix tokens
  - $t_1, t_2, \ldots, t_i$ that the parser identifies as legal:
    - there exists tokens $t_{i+1}, t_{i+2}, \ldots, t_n$ such that $t_1, t_2, \ldots, t_n$ is a syntactically valid program

- If every token is considered as single character:
  - For every prefix word u that the parser identifies as legal there exists w such that u.w is a valid program

Recovery is tricky

- Heuristics for dropping tokens, skipping to semicolon, etc.

Building the Parse Tree

Adding semantic actions

- Can add an action to perform on each production rule
- Can build the parse tree
  - Every function returns an object of type Node
  - Every Node maintains a list of children
  - Function calls can add new children

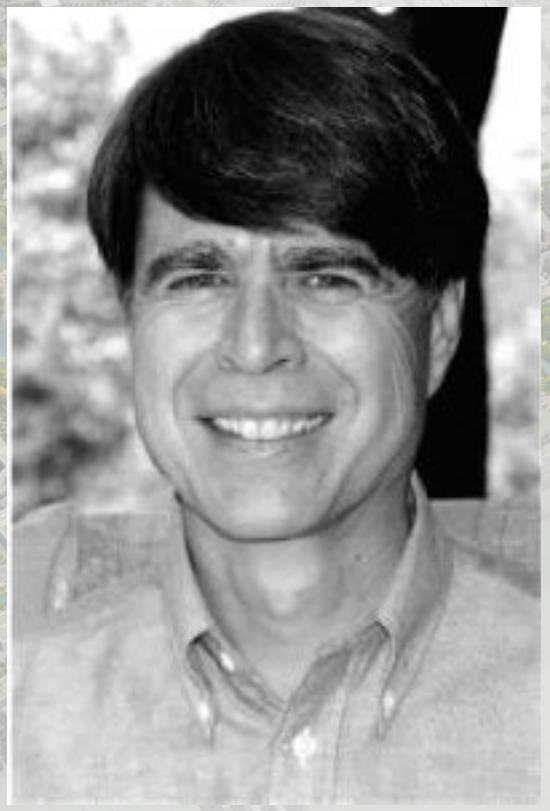
Building the parse tree

```
Node E() {
  result = new Node();
  result.name = "E";
  if (current ∈ {TRUE, FALSE}) // E → LIT
    result.addChild(LIT());
  else if (current == LPAREN)  // E → ( E OP E )
    result.addChild(match(LPAREN));
    result.addChild(E());
    result.addChild(OP());
    result.addChild(E());
    result.addChild(match(RPAREN));
  else if (current == NOT)          // E → not E
    result.addChild(match(NOT));
    result.addChild(E());
  else error;
    return result;
}
```

Parser for Fully Parenthesized Expers

```
static int Parse_Expression(Expression **expr_p) {
  Expression *expr = *expr_p = new_expression() ;
  /* try to parse a digit */
  if (Token.class == DIGIT) {
        expr->type='D';   expr->value=Token.repr -'0';
        get_next_token();
        return 1;       }
/* try parse parenthesized expression */
if (Token.class == '(') {
      expr->type='P';    get_next_token();
      if (!Parse_Expression(&expr->left))   Error("missing expression");
      if (!Parse_Operator(&expr->oper))   Error("missing operator");
      if (Token.class != ')') Error("missing )");
      get_next_token();
      return 1; }
return 0;
}
```

Earley Parsing



Jay Earley, PhD

Earley Parsing

- Invented by Jay Earley [PhD. 1968]
- Handles arbitrary context free grammars
  - Can handle ambiguous grammars
- Complexity $O(N^3)$ when N = |input|
- Uses dynamic programming
  - Compactly encodes ambiguity

Dynamic programming

- Break a problem P into subproblems $P_1 \ldots P_k$
  - Solve P by combining solutions for $P_1 \ldots P_k$
  - Memo-ize (store) solutions to subproblems instead of re-computation
- Bellman-Ford shortest path algorithm
  - Sol(x,y,i) = minimum of
    - Sol(x,y,i-1)
    - Sol(t,y,i-1) + weight(x,t) for edges (x,t)

Earley Parsing

- Dynamic programming implementation of a recursive descent parser
  - S[N+1] Sequence of sets of "Earley states"
    - N = |INPUT|
    - Earley state (item) s is a sentential form + aux info
  - S[i] All parse tree that can be produced (by a RDP) after reading the first i tokens
    - S[i+1] built using S[0] … S[i]

Earley Parsing

- Parse arbitrary grammars in $O(|input|^3)$
  - $O(|input|^2)$ for unambigous grammer
  - Linear for most LR(k) langaues (next lesson)
- Dynamic programming implementation of a recursive descent parser
  - S[N+1] Sequence of sets of “Earley states”
    - N = |INPUT|
    - Earley states is a sentential form + aux info
  - S[i] All parse tree that can be produced (by an RDP) after reading the first i tokens
    - S[i+1] built using S[0] … S[i]

Earley States

- s = < constituent, back >
  - constituent (dotted rule) for A→αβ
    - A→•αβ predicated constituents
    - A→α•β in-progress constituents
    - A→αβ• completed constituents
  - back previous Early state in derivation

Earley States

- s = < constituent, back >
  - constituent (dotted rule) for A→αβ
    - A→•αβ predicated constituents
    - A→α•β in-progress constituents
    - A→αβ• completed constituents
  - back previous Early state in derivation

Earley Parser

Input = x[1…N]

S[0] = <E'→ •E, 0>; S[1] = … S[N] = {}

for i = 0 ... N do

  until S[i] does not change do

    foreach s ∈ S[i]

      if s = <A→…•a…, b> and a=x[i+1] then

        S[i+1] = S[i+1] ∪ {<A→…a•…, b> } // scan

      if s = <A→ … •X …, b> and X→α then

        S[i] = S[i] ∪ {<X→•α, i > } // predict

      if s = < A→ … • , b> and <X→…•A…, k> ∈ S[b] then

        S[i] = S[i] ∪{<X→…A•…, k> } // complete

Earley Parser

Input = x[1…N]

S[0] = <E'→ •E, 0>; S[1] = … S[N] = {}

for i = 0 ... N do

  until S[i] does not change do

    foreach s ∈ S[i]

---

if s = <A→…•a…, b> and a=x[i+1] then // scan

  S[i+1] = S[i+1] ∪ {<A→…a•…, b> }

---

if s = <A→ … •X …, b> and X→α then // predict

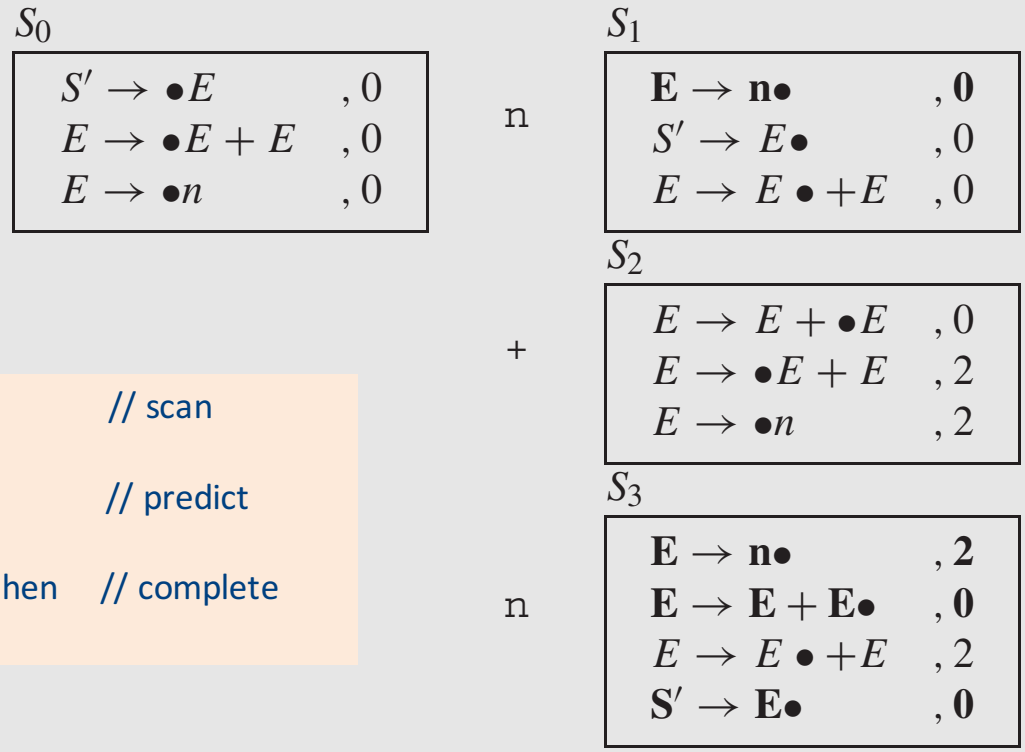
  S[i] = S[i] ∪ {<X→•α, i > }

---

if s = < A→ … • , b> and <X→…•A…, k> ∈ S[b] then // complete

  S[i] = S[i] ∪{<X→…A•…, k> }

Example



if s = <A→…•a…, b> and a=x[i+1] then // scan
S[i+1] = S[i+1] ∪ {<A→…a•…, b> }
if s = <A→ … •X …, b> and X→α then // predict
S[i] = S[i] ∪ {<X→•α, i > }
if s = < A→ … • , b> and <X→…•A…, k> ∈ S[b] then // complete
S[i] = S[i] ∪{<X→…A•…, k> }

FIGURE 1. Earley sets for the grammar $E \rightarrow E + E \mid n$ and the input `n + n`. Items in bold are ones which correspond to the input's derivation.

Earley Parser

Input = x[1…N]

S[0] = <E'→ •E, 0>; S[1] = … S[N] = {}

for i = 0 ... N do

  until S[i] does not change do

    foreach s ∈ S[i]

---

if s = <A→…•a…, b> and a=x[i+1] then // scan

  S[i+1] = S[i+1] ∪ {<A→…a•…, b> }

---

if s = <A→ … •X …, b> and X→α then // predict

  S[i] = S[i] ∪ {<X→•α, i > }

---

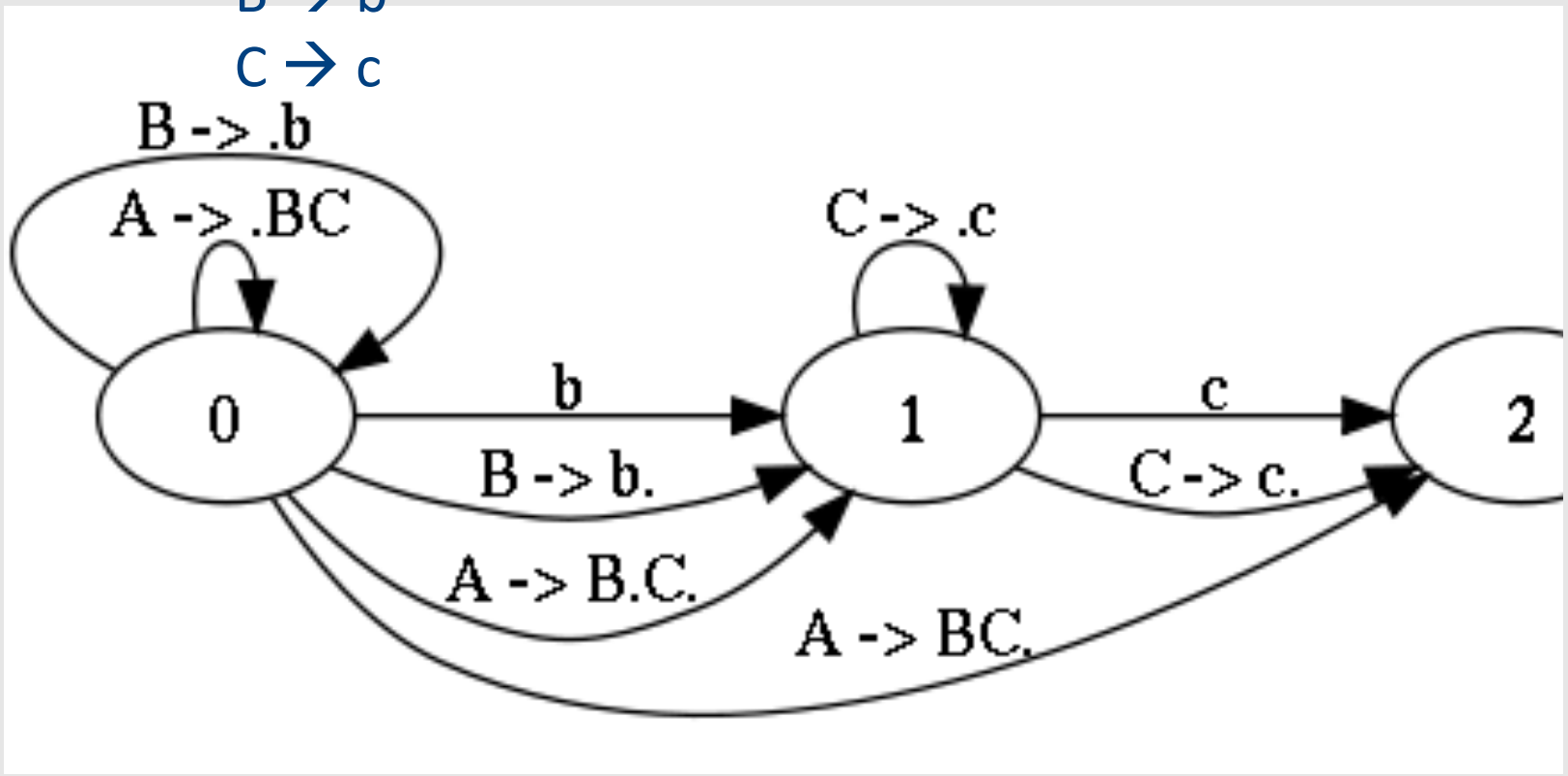
if s = < A→ … • , b> and <X→…•A…, k> ∈ S[b] then // complete

  S[i] = S[i] ∪{<X→…A•…, k> }

Earley with Pictures

Grammar: A → BC
B → b
C → c

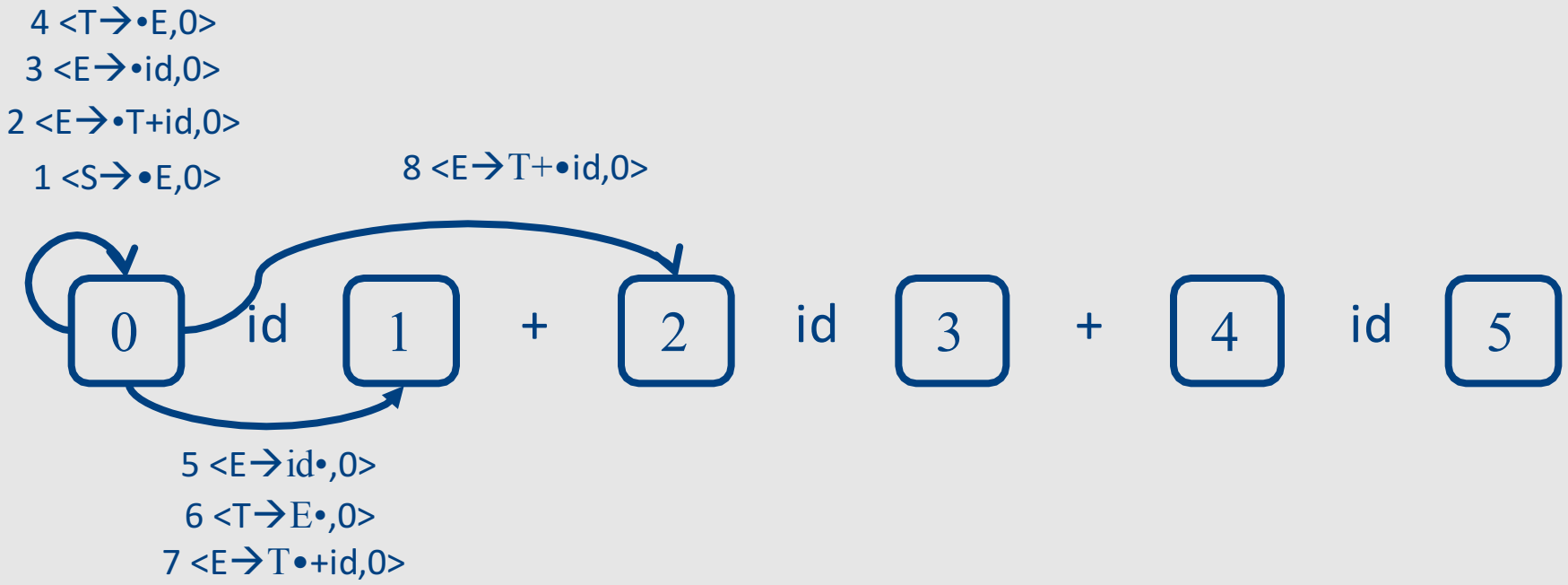
Input: bc



Earley Parsing in Pictures

Grammar: S → E
E → T + id | id
T → E

Input: id + id + id




That's all Folks