Compilation

0368-3133 (Semester A, 2013/14)

Lecture 4: Syntax Analysis
(Top-Down Parsing)

Modern Compiler Design: Chapter 2.2

Noam Rinetzky

Slides credit: Roman Manevich, Mooly Sagiv, Jeff Ullman, Eran Yahav

Admin

- Next week: Trubowicz 101 (Law school)

- Mobiles ...

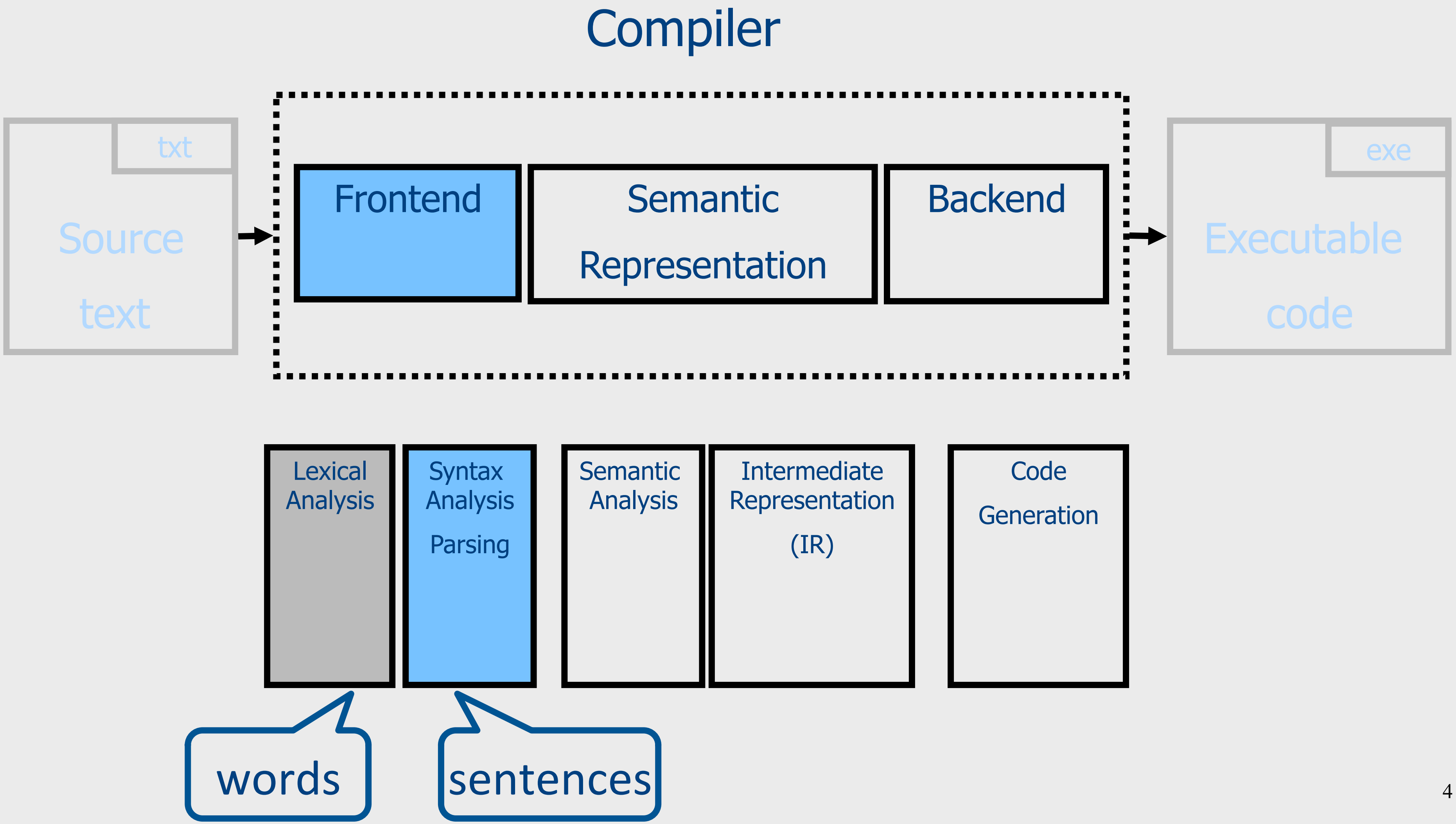
What is a Compiler?

“A compiler is a computer program that transforms source code written in a programming language (source language) into another language (target language).

The most common reason for wanting to transform source code is to create an executable program.”

--*Wikipedia*

Conceptual Structure of a Compiler



Compiler

txt

Source

text

Frontend

Semantic

Representation

Backend

exe

Executable

code

Lexical Analysis

Syntax Analysis

Parsing

Semantic Analysis

Intermediate Representation

(IR)

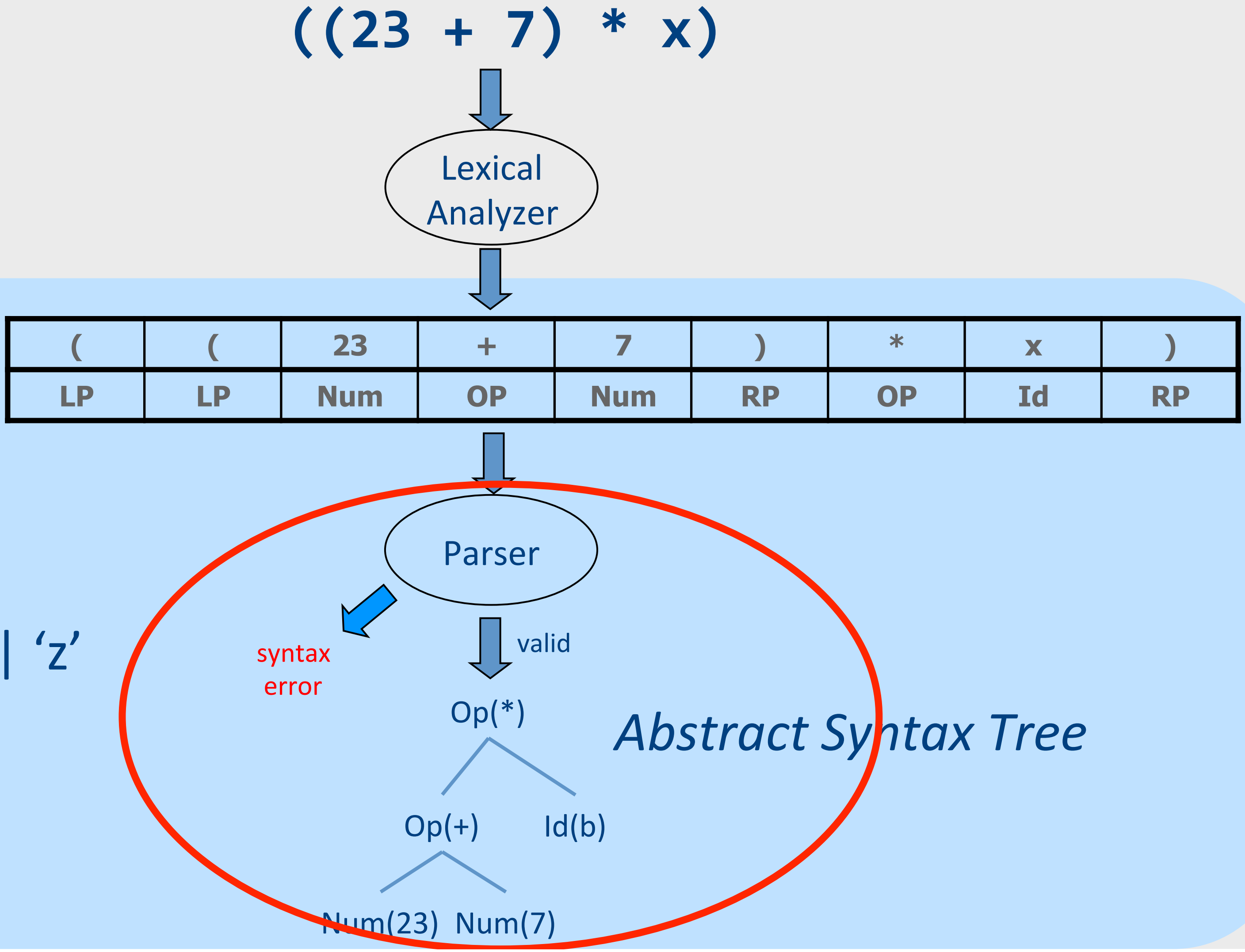
Code

Generation

words

sentences

From scanning to parsing



program text

`((23 + 7) * x)`

token stream

(	(	23	+	7	)	*	x	)
LP	**LP**	**Num**	**OP**	**Num**	**RP**	**OP**	**Id**	**RP**

Grammar:

E → … | Id

Id → ‘a’ | … | ‘z’

Abstract Syntax Tree

Context Free Grammars

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

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

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 in which there may be non-terminals

Parse Tree

S

S ; S

id := E; S

id := id; S

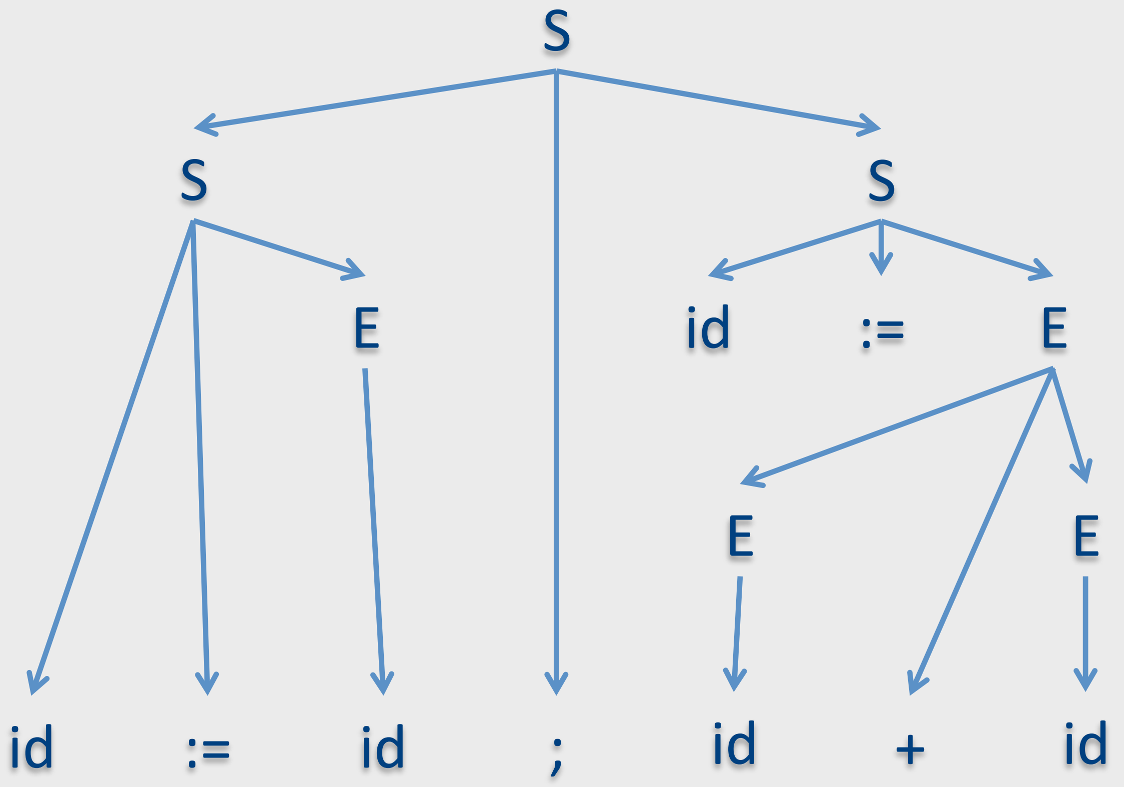
id := id;id := E

id := id; id := E + E

id := id;id := E + id

id := id;id := id + id

x:= z ; y := x + z



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

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
- Linear 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

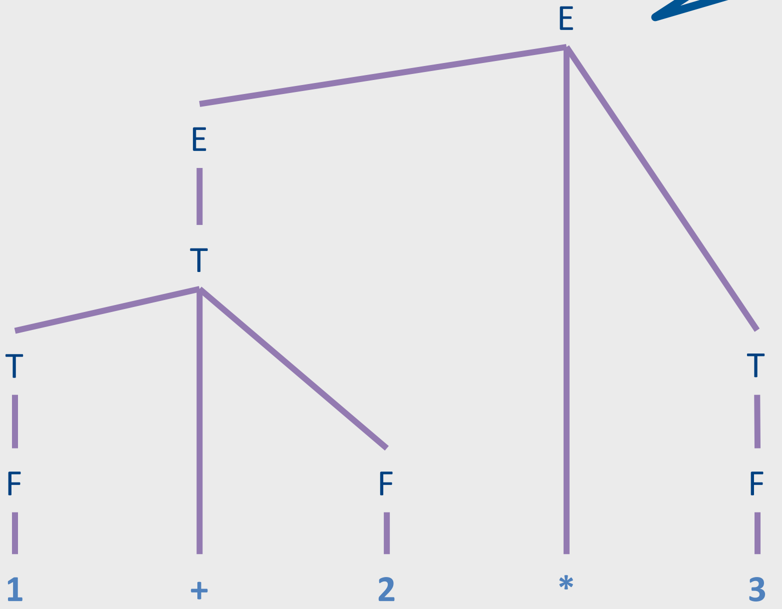
T → F

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)

We need this rule to get the *

E

1 + 2 * 3

Intuition: Top-Down parsing

Unambiguous grammar

E → E * T

E → T

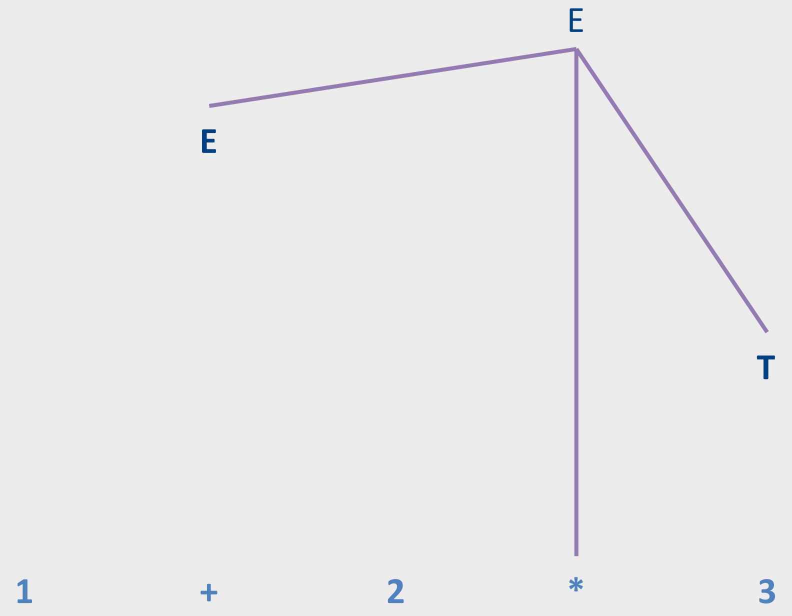
T → T + F

T → F

F → id

F → num

F → (E)



Intuition: Top-Down parsing

Unambiguous grammar

E → E * T

E → T

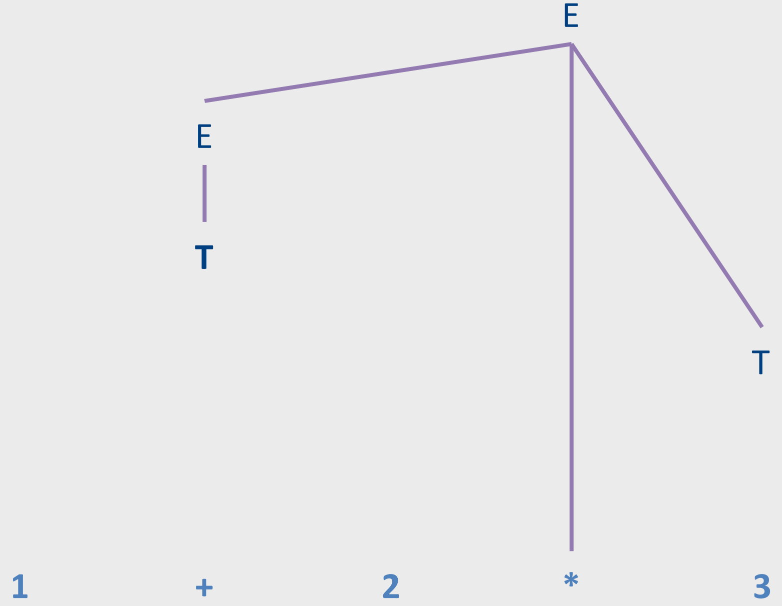
T → T + F

T → F

F → id

F → num

F → (E)



Intuition: Top-Down parsing

Unambiguous grammar

E → E * T

E → T

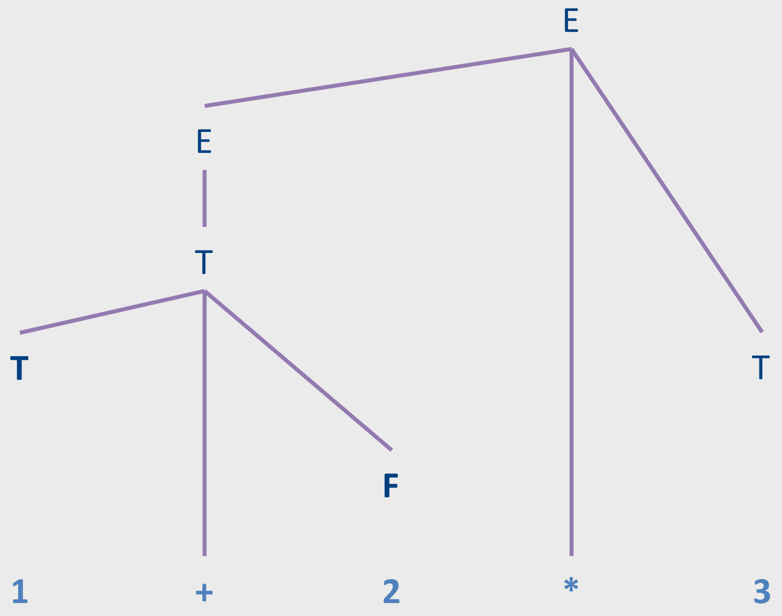
T → T + F

T → F

F → id

F → num

F → (E)



Intuition: Top-Down parsing

Unambiguous grammar

E → E * T

E → T

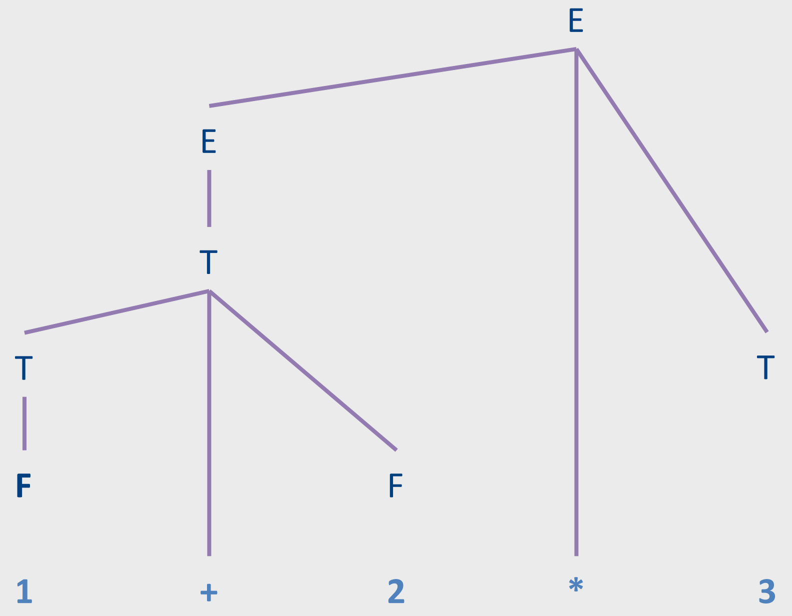
T → T + F

T → F

F → id

F → num

F → (E)



Intuition: Top-Down parsing

Unambiguous grammar

E → E * T

E → T

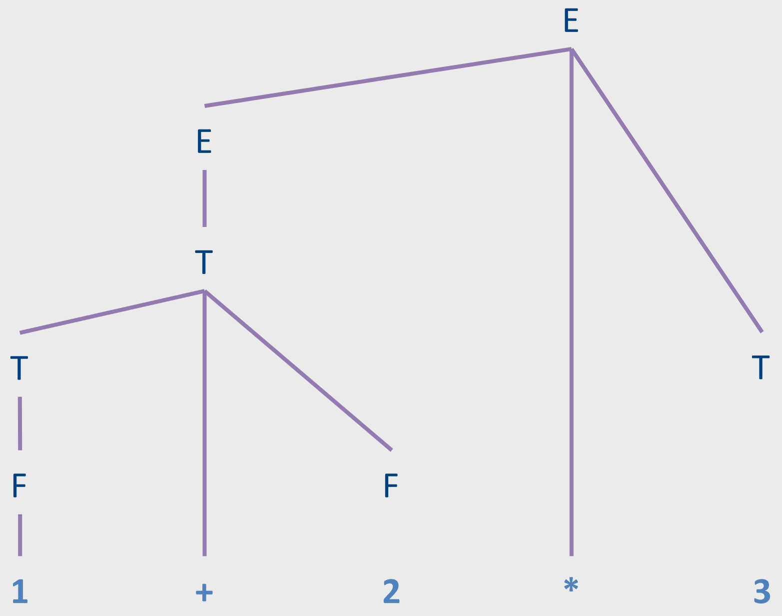
T → T + F

T → F

F → id

F → num

F → (E)



Intuition: Top-Down parsing

Unambiguous grammar

E → E * T

E → T

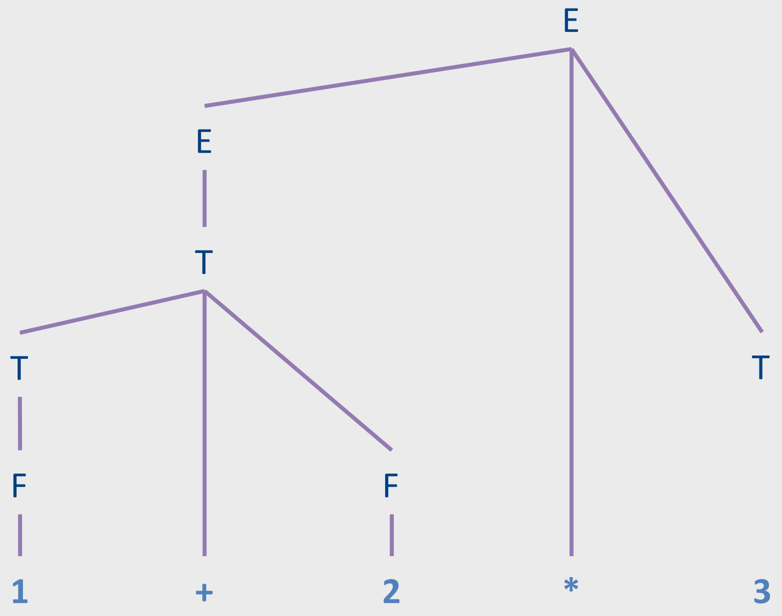
T → T + F

T → F

F → id

F → num

F → (E)



Intuition: Top-Down parsing

Unambiguous grammar

E → E * T

E → T

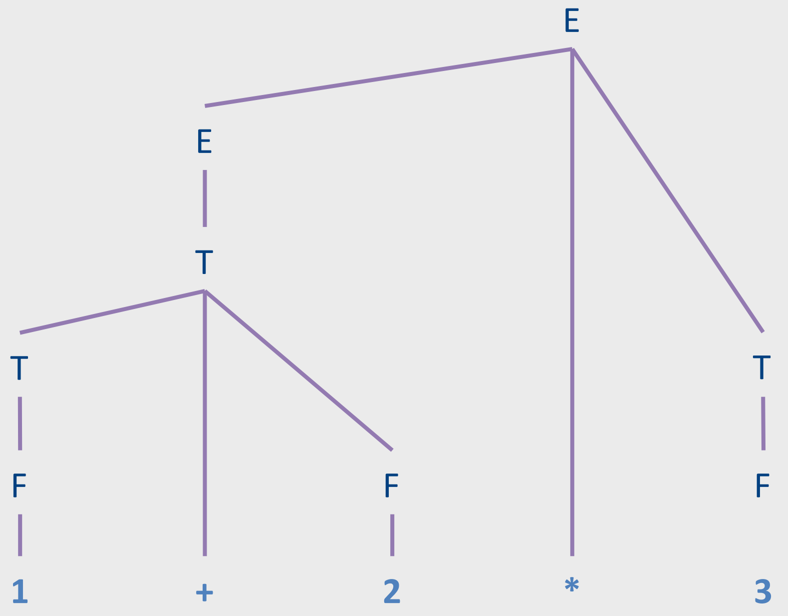
T → T + F

T → F

F → id

F → num

F → (E)



Intuition: Top-Down parsing

Unambiguous grammar

E → E * T

E → T

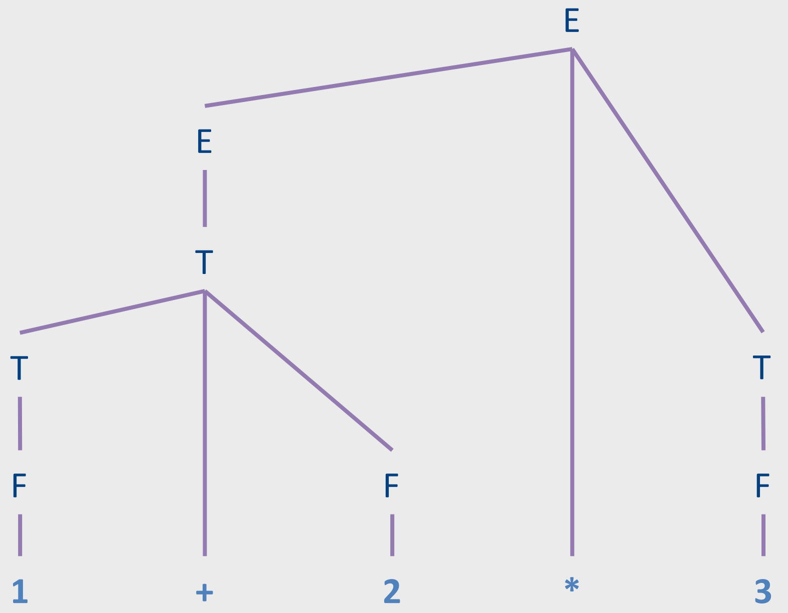
T → T + F

T → F

F → id

F → num

F → (E)



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?

Recursive Descent

- Blind exhaustive search
  - Goes over all possible production rules
  - Read & parse prefixes of input
  - Backtracks if guesses wrong

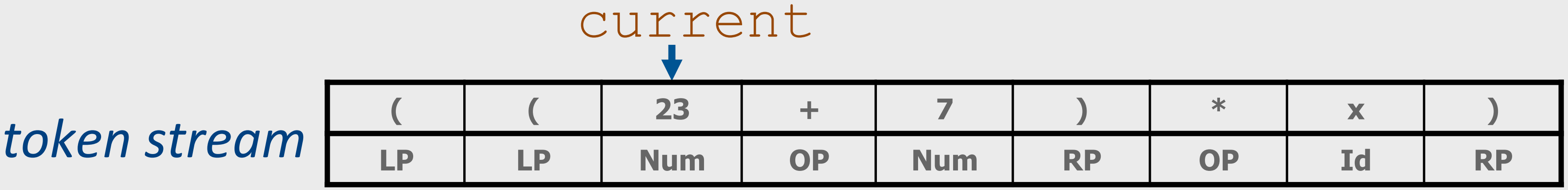
- Implementation
  - Uses (possibly recursive) functions for every production rule
  - Backtracks ➔ "rewind" input

Recursive descent

```
bool A(){ // A → $A_1$ | … | $A_n$
  pos = recordCurrentPosition();

  for (i = 1; i ≤ n; i++){
    if ($A_i$())
      return true;
    rewindCurrent(pos);
  }
  return false;
}

bool $A_i$(){  // $A_i$ = $X_1X_2…X_k$
  for (j=1; j ≤ k; j++)
    if ($X_j$ is a terminal)
      if ($X_j$ == current) match(current);
      else return false;
    else if (! $X_j$()) return false;
  return true;
}
```



`current`

token stream

(	(	23	+	7	)	*	x	)
LP	**LP**	**Num**	**OP**	**Num**	**RP**	**OP**	**Id**	**RP**

Example

- Grammar
  - E → T | T + E
  - T → int | int * T | (E)

- Input: (5)
- Token stream: LP int RP

p. 127

Problem: Left Recursion

E → E - term | term

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

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

p. 130

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:

$E \rightarrow E$ - term | term

$E \rightarrow$ term TE | term
$TE \rightarrow$ - term TE | ε

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?
- Wanted: Top-Down parsing without backtracking

Predictive parsing

- Given a grammar G and a word w derive w using G
    - 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 based on
        - Non terminal
        - Next (k) tokens (lookahead)

Boolean expressions example

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

not (not true or false)

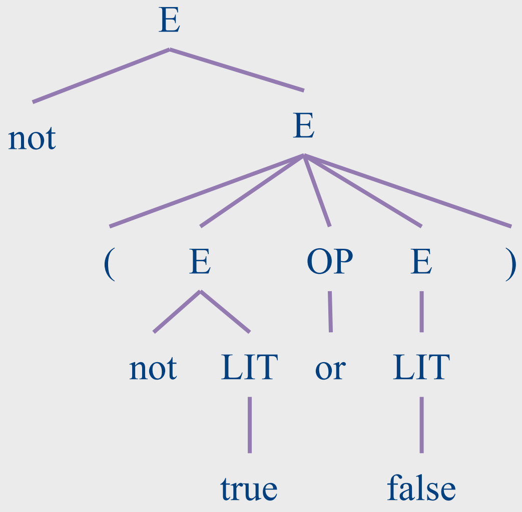
Boolean expressions example

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

not (not true or false)

production to apply known from next token

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)



Problem: productions with common prefix

term → ID | ID [expr]

- Cannot tell which rule to use based on lookahead (ID)

Solution: left factoring

- Rewrite the grammar to be in LL(1)

term → ID | 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

LL(k) Parsers

- Predictive parser
  - Can be generated automatically
  - Does not use recursion
  - Efficient

- In contrast, recursive descent
  - Manual construction
  - Recursive
  - Expensive

LL(k) parsing via pushdown automata and prediction table

- Pushdown automaton uses
  - Prediction stack
  - Input token 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

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

Which rule should be used

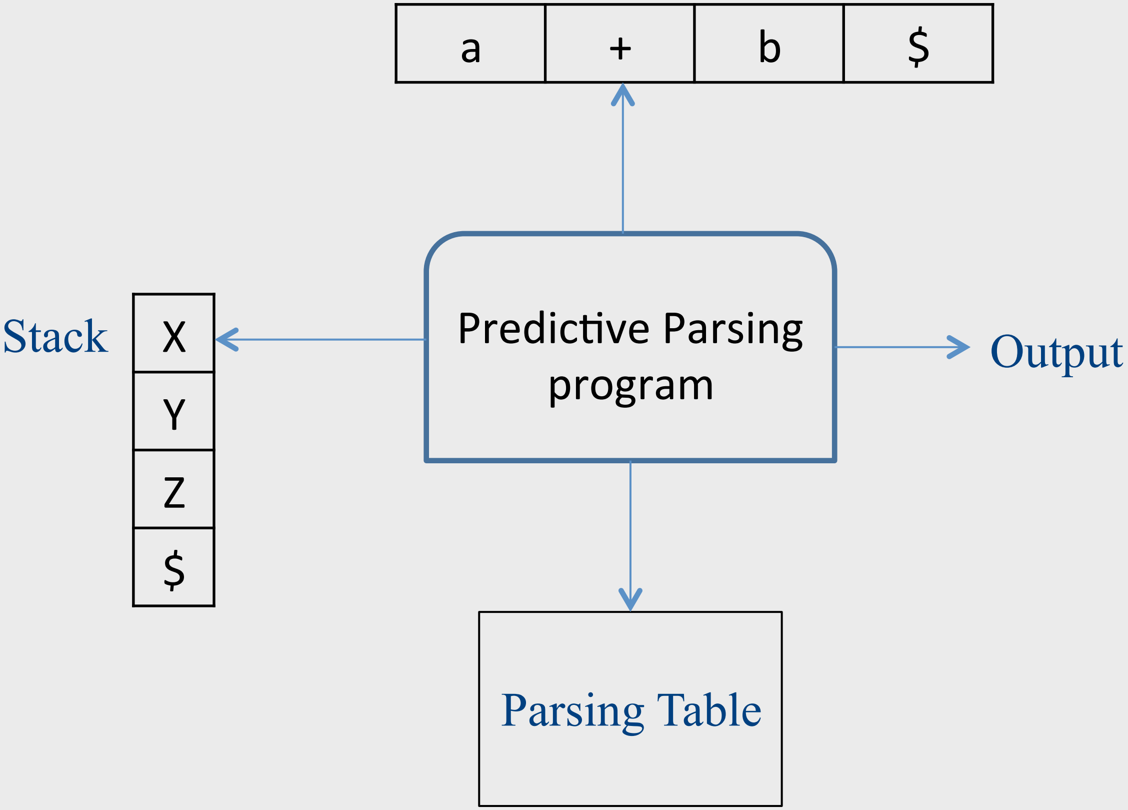
Input tokens

Nonterminals

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

Table

Non-Recursive Predictive Parser



a	+	b	$

Stack

X
Y
Z
$

Predictive Parsing program

Output

Parsing Table

LL(k) parsing via pushdown automata and prediction table

- 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

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

FIRST sets

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

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

Computing FIRST sets

- FIRST (t) = { t }   // “t” non terminal

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

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

FIRST sets computation example

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

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

2. Iterate 2

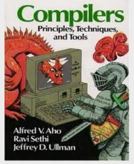
STMT → if EXPR then STMT
| while EXPR do STMT
| EXPR ;
EXPR → TERM -> id
| zero? TERM
| not EXPR
| ++ id
| -- id
TERM → id
| constant

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
| 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

- FOLLOW(X) = { t | S$\rightarrow^*$ α X t β}
  - FOLLOW(X) = set of tokens that can immediately follow X in some sentential form
  - NOT related to what can be derived from X

- Intuition: X → A B
  - FIRST(B) ⊆ FOLLOW(A)
  - FOLLOW(B) ⊆ FOLLOW(X)
  - If B$\rightarrow^*\varepsilon$ then FOLLOW(A) ⊆ FOLLOW(X)

FOLLOW sets: Constraints

- $\$ \in \text{FOLLOW}(S)$

- $\text{FIRST}(\beta) - \{\varepsilon\} \subseteq \text{FOLLOW}(X)$
  - For each $A \rightarrow \alpha\ X\ \beta$

- $\text{FOLLOW}(A) \subseteq \text{FOLLOW}(X)$
  - For each $A \rightarrow \alpha\ X\ \beta$ and $\varepsilon \in \text{FIRST}(\beta)$

Example: FOLLOW sets

- E→ TX   X→+ E | Ɛ
- T→ (E) | int Y   Y → * T | Ɛ

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

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

Prediction Table

- A → α

- T[A,t] = α   if t ∈ FIRST(α)
- T[A,t] = α   if ε ∈ FIRST(α) and t ∈ FOLLOW(A)
  - t can also be $

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

Problem: Non LL Grammars

$S \rightarrow A\ a\ b$
$A \rightarrow a \mid \varepsilon$

```
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

$S \rightarrow A\ a\ b$
$A \rightarrow a \mid \varepsilon$

- 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

LL(k) grammars

- A grammar is LL(k) if 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)
  - T is well defined
- A language is said to be LL(k) when it has an LL(k) grammar

LL(k) grammars

- A grammar is not LL(k) if it is
  - Ambiguous
  - Left recursive
  - Not left factored
  - …

Earley Parsing

- Invented by Earley [PhD. 1968]

- Handles arbitrary CFG
- 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]

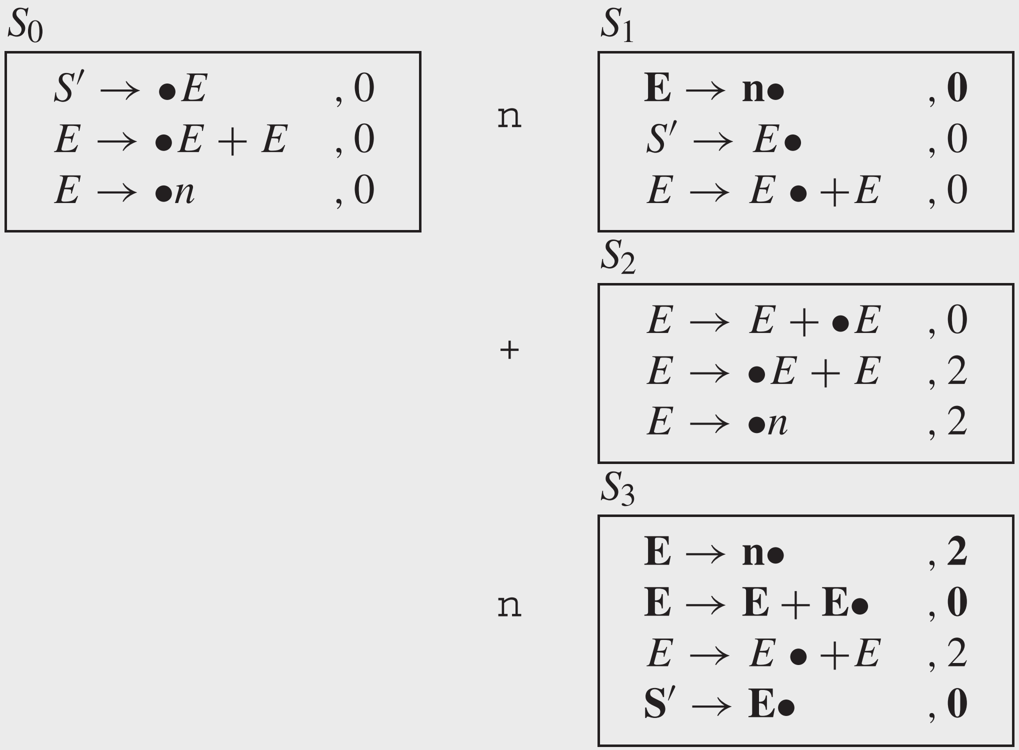
Earley States

- 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
```

Example



S_0

$S' \to \bullet E \quad , 0$

$E \to \bullet E + E \quad , 0$

$E \to \bullet n \quad , 0$

n

S_1

$\mathbf{E \to n\bullet} \quad \mathbf{, 0}$

$S' \to E\bullet \quad , 0$

$E \to E \bullet + E \quad , 0$

+

S_2

$E \to E + \bullet E \quad , 0$

$E \to \bullet E + E \quad , 2$

$E \to \bullet n \quad , 2$

n

S_3

$\mathbf{E \to n\bullet} \quad \mathbf{, 2}$

$\mathbf{E \to E + E\bullet} \quad \mathbf{, 0}$

$E \to E \bullet + E \quad , 2$

$\mathbf{S' \to E\bullet} \quad \mathbf{, 0}$

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