Compilation

0368-3133

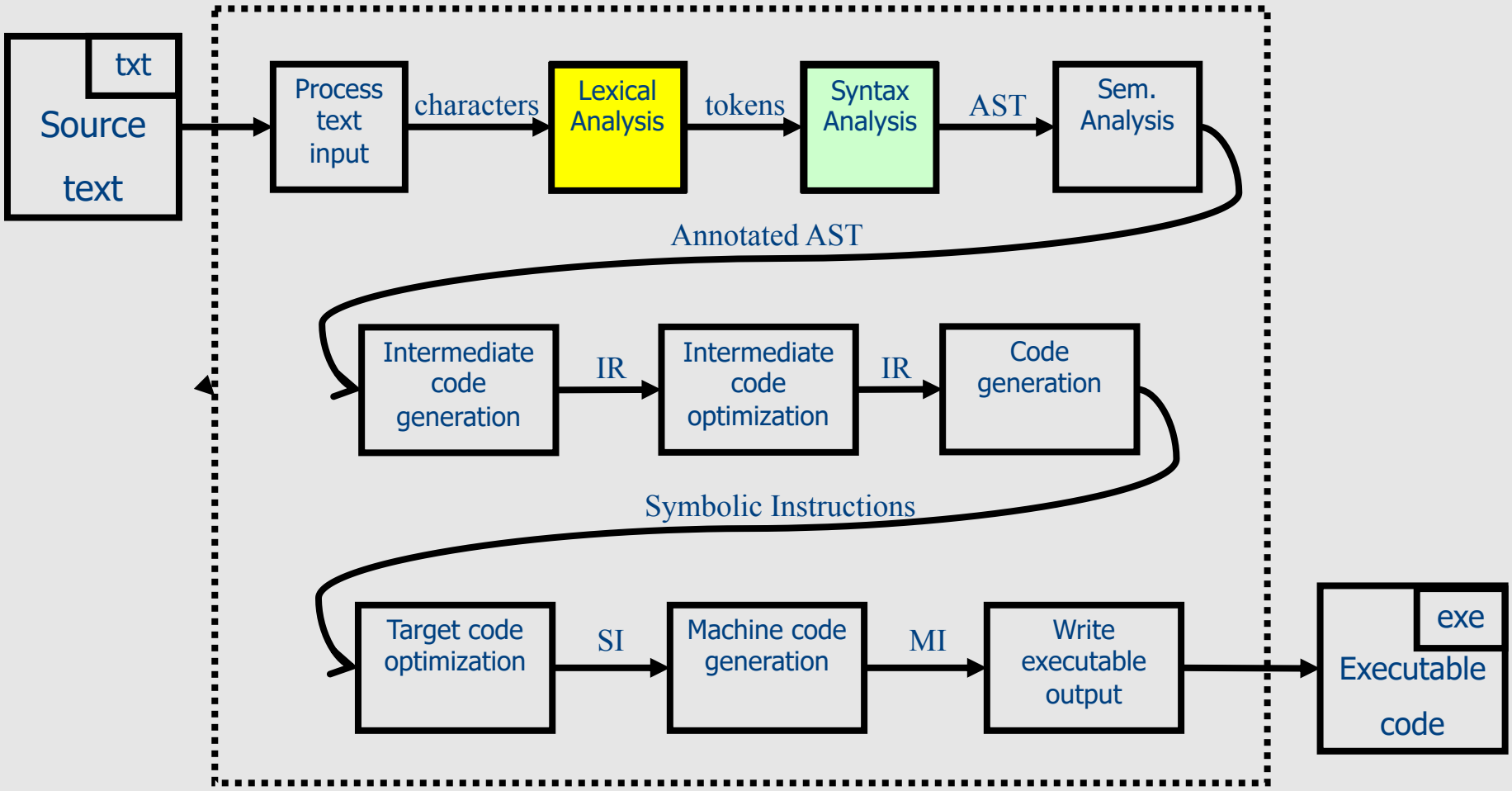
Lecture 3a:

Syntax Analysis:

Top-Down parsing

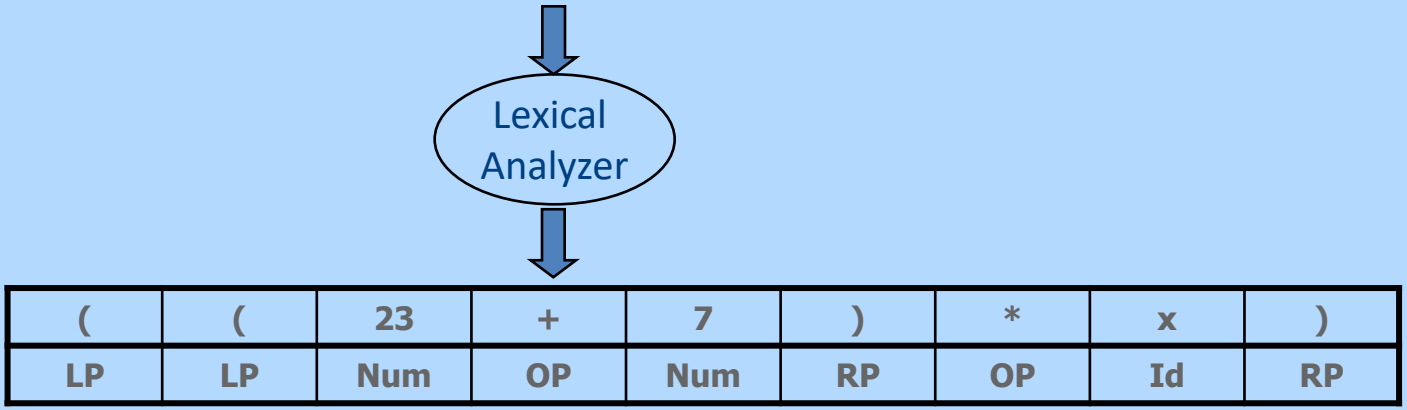
Noam Rinetzky

The Real Anatomy of a Compiler



Frontend: Scanning & Parsing

program text ((23 + 7) * x)



Grammar:

E → ... | Id

Id → ‘a’ | ... | ‘z’

Abstract Syntax Tree

From scanning to parsing

program text

```
((23 + 7) * x)
```

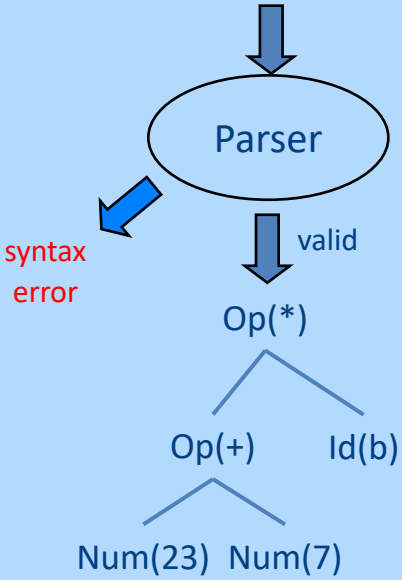
token stream

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

Grammar:

E → ... | Id

Id → ‘a’ | ... | ‘z’



Abstract Syntax Tree

Reminder

- Context-free languages
  - Grammars
  - Pushdown Automata
- Terminology
  - Derivation
  - Sentential form
  - Parse trees
  - Leftmost/rightmost derivation
  - Ambiguous grammars
- Top-down / Botto-up parsers

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?

Top-down parsing

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)

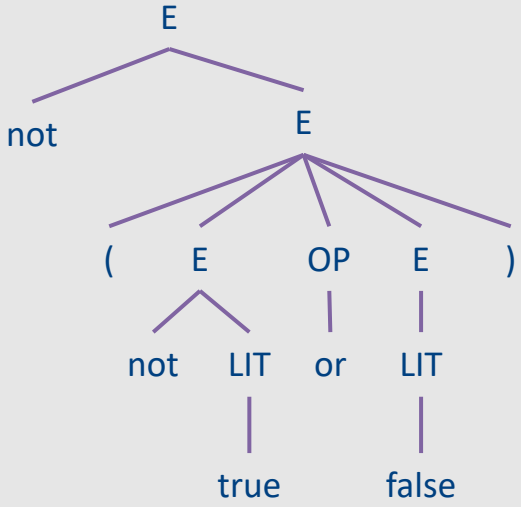
Boolean expressions example

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

not (not true or false)

production to apply known from next token

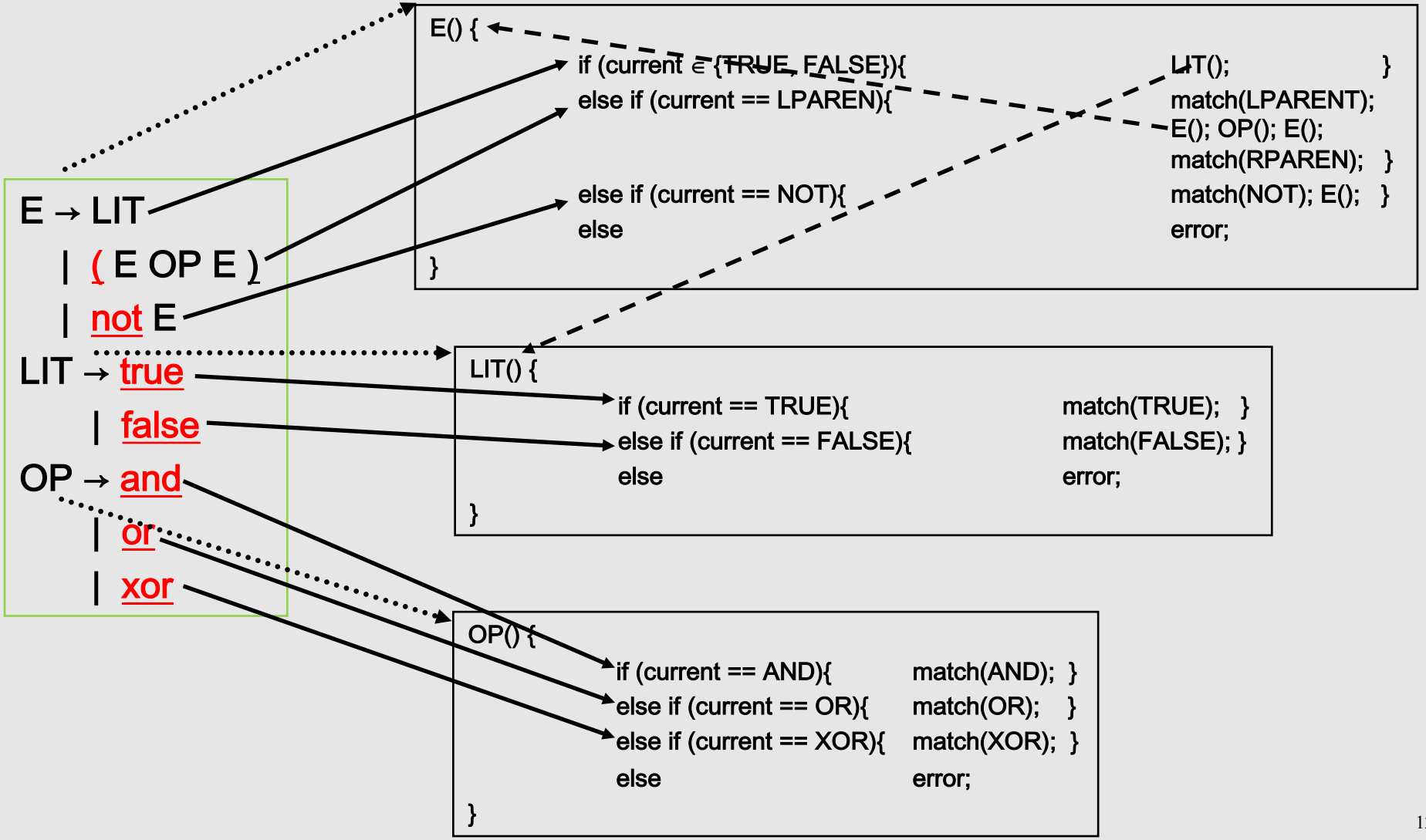
E =>
not E =>
not (E OP E) =>
not (not E OP E) =>
not (not LIT OP E) =>
not (not true OP E) =>
not (not true or E) =>
not (not true or LIT) =>
not (not true or false)



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

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]`**

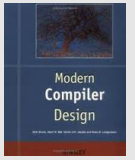
Problem 2: null productions

$S \rightarrow A\ a\ b$

$A \rightarrow a \mid \varepsilon$

```
S() {
  return  A()  ;  match(token('a'))  ;  match(token('b'))
}
A()  {
  match(token('a'))   ||   skip
}
```

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



Problem 3: left recursion

$E \rightarrow E$ - term | term

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

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

What can we do?

FIRST sets

X → YY | Z Z | Y Z | 1 Y

Y → 4 | ε

Z → 2

L(Z) = {2}

L(Y) = {4, ε}

L(X) = {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(X) = {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 $A \rightarrow \varepsilon$
  1. Initially, for all nonterminals A, set
     FIRST(A) = { t | $A \rightarrow t\omega$ *for some* ω }
  2. Repeat the following until no changes occur:
     for each nonterminal A
       for each production $A \rightarrow B\omega$
         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

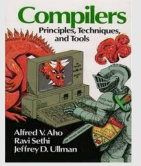
STMT → if EXPR then STMT
| while EXPR do STMT
| EXPR ;
EXPR → TERM -> id
| zero? TERM
| not EXPR
| ++ id
| -- id
TERM → id
| constant

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?
  - $A \rightarrow B\ C\ D \quad B \rightarrow \varepsilon\ C \rightarrow \varepsilon$
  - Use what comes afterwards to predict the right production
- For every production rule $A \rightarrow \alpha$
  - 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) $\rightarrow$ (if A_k is nullable then FOLLOW(N))

FOLLOW sets: Constraints

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

Example: FOLLOW sets

- E→ TX
- T→ (E) | int Y

X→+ E | Ɛ

Y → * T | Ɛ

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(1) 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

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

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

$S \rightarrow$ if E then S else S
| if E then S
| T

$S \rightarrow$ if E then S S'
| T
$S' \rightarrow$ else S | ε

Back to problem 2

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

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

- FIRST/FOLLOW conflict

Solution: substitution

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

Substitute A in S

$S \rightarrow a\ a\ b \mid a\ b$

Left factoring

$S \rightarrow a\ after\_A$
$after\_A \rightarrow a\ b \mid b$

Back to problem 3

$E \rightarrow E$ - 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:

$E \rightarrow E$ - term | term

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

LL(k) Parsers

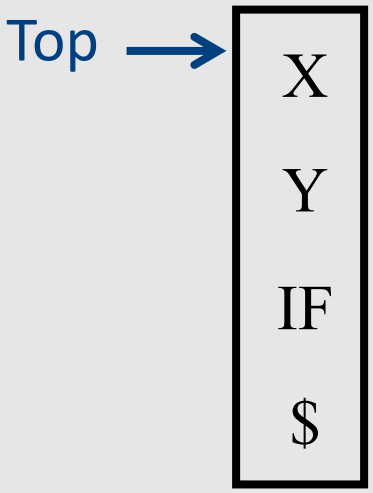
- Recursive Descent
  - Manual construction
  - Uses recursion

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

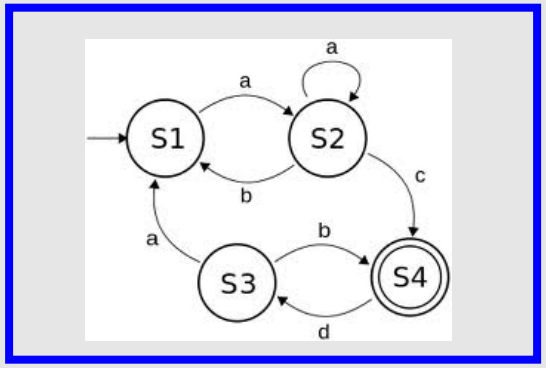
Pushdown Automata (PDA)

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

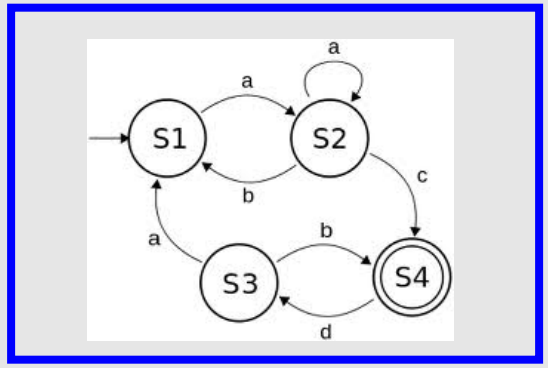
Intuition: PDA

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.

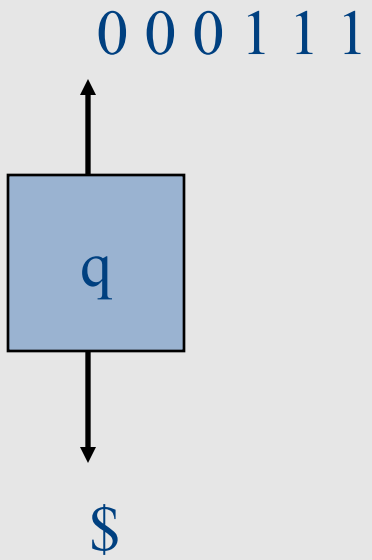
Example: Stack Symbols

- $ = 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

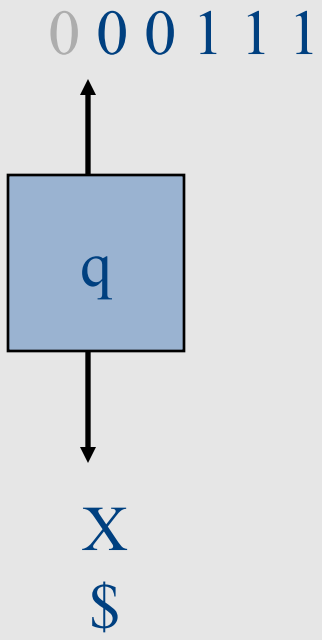
Example: Transitions

- $\delta(q, 0, \$) = \{(q, X\$)\}$.
- $\delta(q, 0, X) = \{(q, XX)\}$.
  - 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.

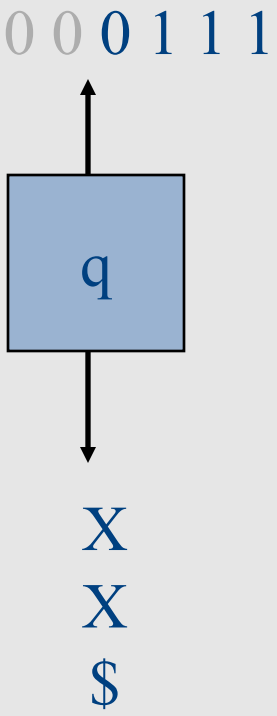
Actions of the Example PDA



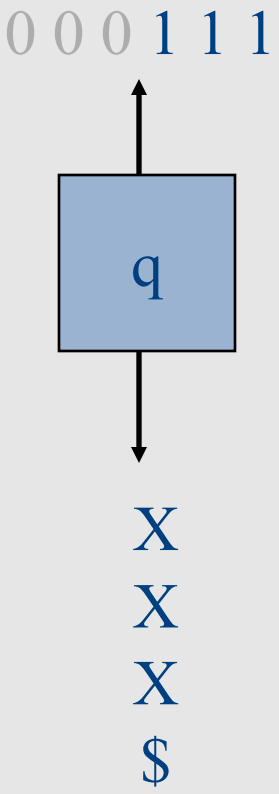
Actions of the Example PDA



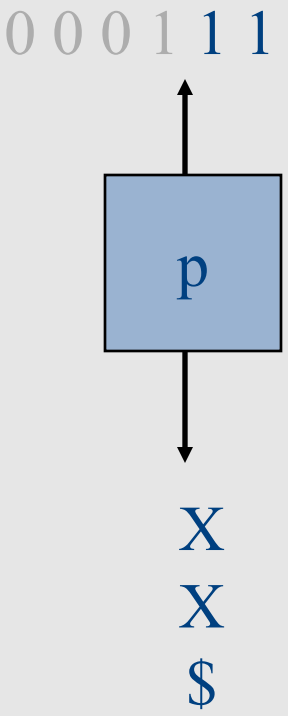
Actions of the Example PDA



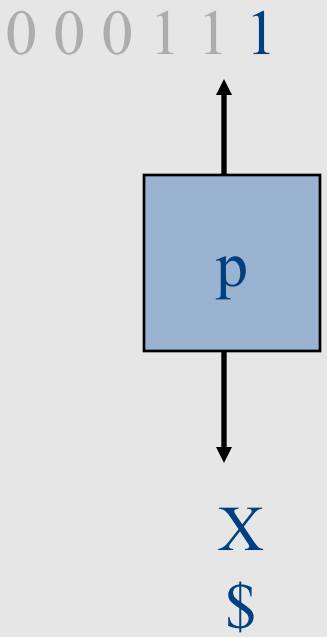
Actions of the Example PDA



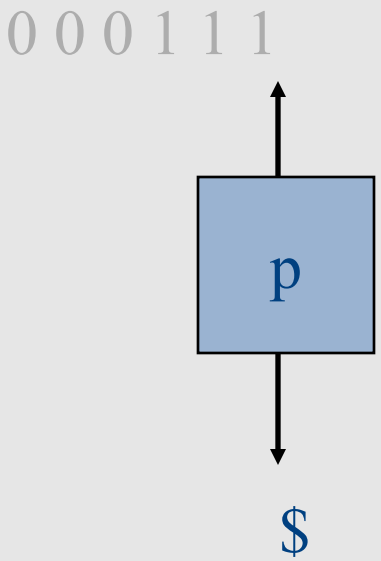
Actions of the Example PDA



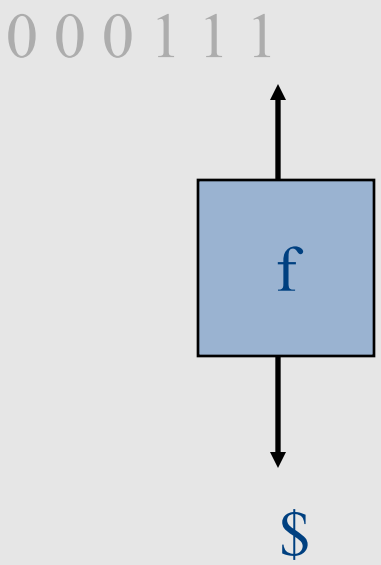
Actions of the Example PDA



Actions of the Example PDA



Actions of the Example PDA



Example: Non Deterministic PDA

- A PDA that accepts palindromes
  - L {pp’ ∈ Σ* | p’=reverse(p)}

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

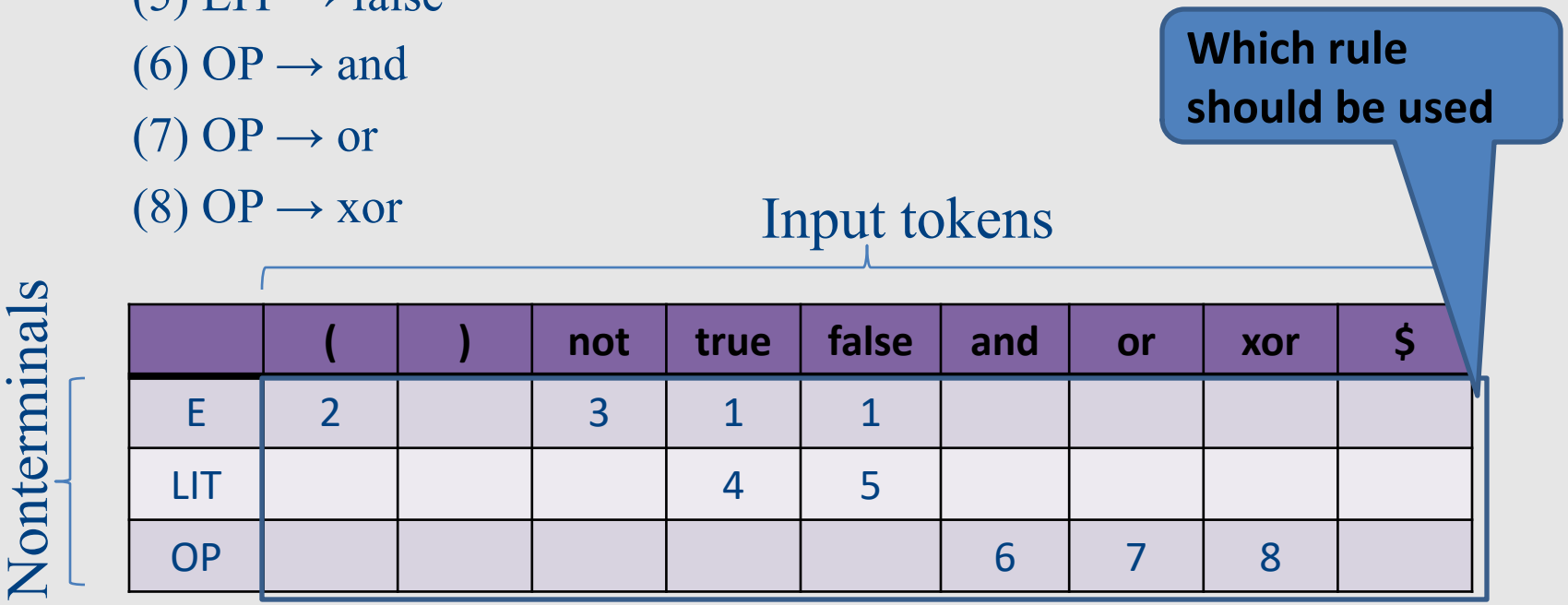
(4) LIT → true

(5) LIT → false

(6) OP → and

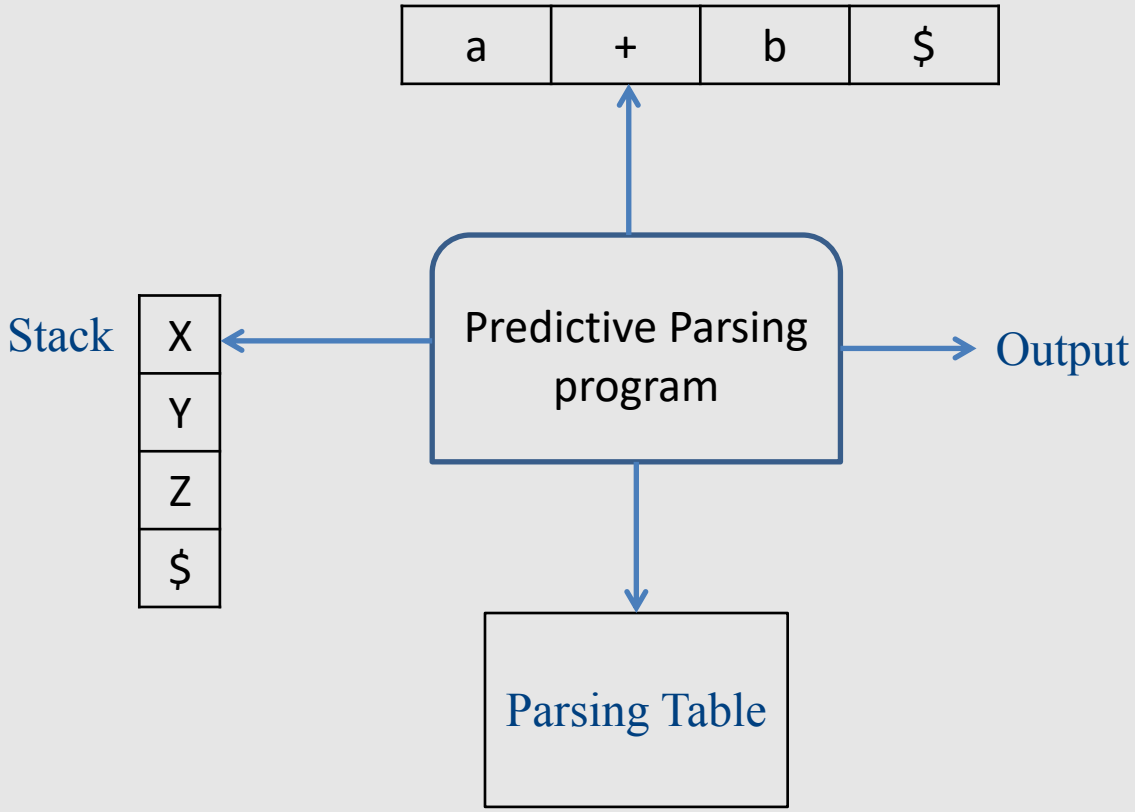
(7) OP → or

(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$

$A \rightarrow aAb \mid c$

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

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

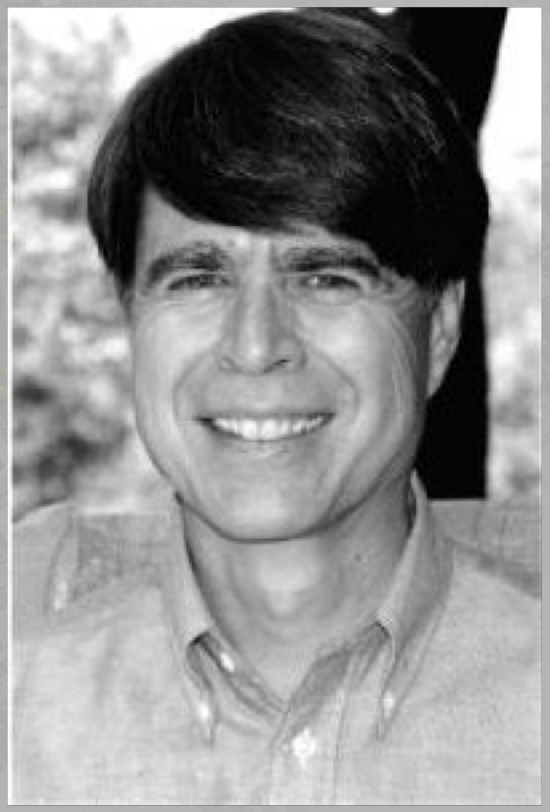
Building the parse tree

```
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 $A \rightarrow \alpha\beta$
    - $A \rightarrow \bullet\alpha\beta$ predicated constituents
    - $A \rightarrow \alpha\bullet\beta$ in-progress constituents
    - $A \rightarrow \alpha\beta\bullet$ completed constituents
  - back previous Early state in derivation

Earley States

- s = < constituent, back >
  - constituent (dotted rule) for $A \rightarrow \alpha\beta$
    - $A \rightarrow \bullet\alpha\beta$ predicated constituents
    - $A \rightarrow \alpha\bullet\beta$ in-progress constituents
    - $A \rightarrow \alpha\beta\bullet$ 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 // scan

  S[i+1] = S[i+1] ∪ {<A→…a•…, b> }

---

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

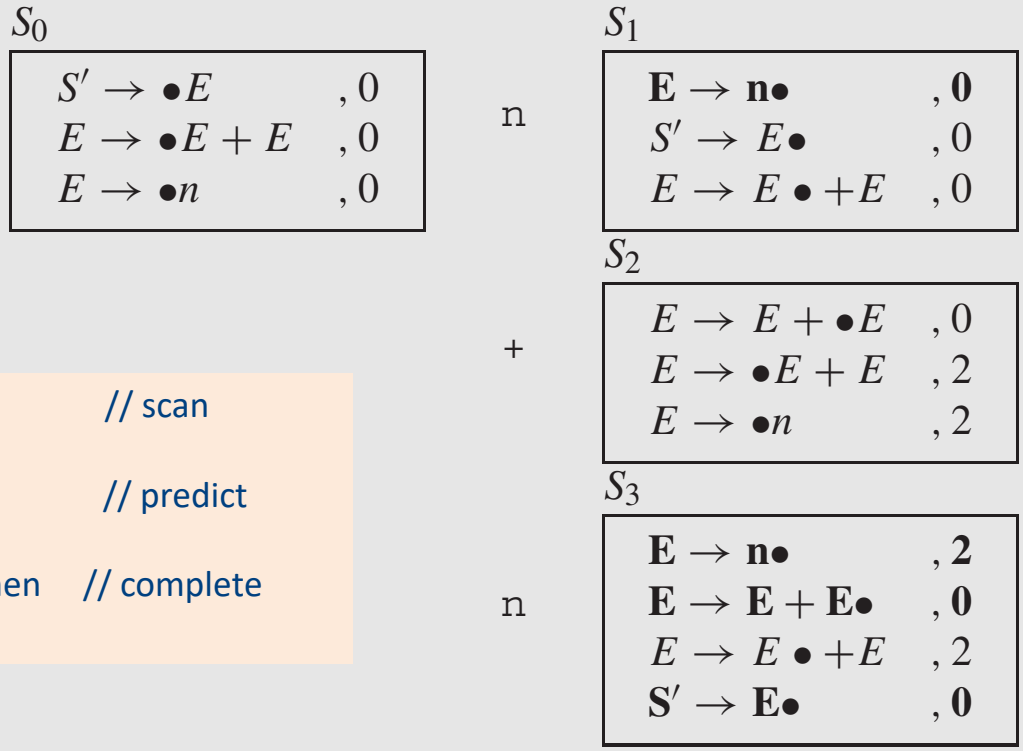
  S[i] = S[i] ∪ {<X→•α, i > }

---

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.