CS480
Translators

Intro to Parsing

Chap. 4

Oregon State University

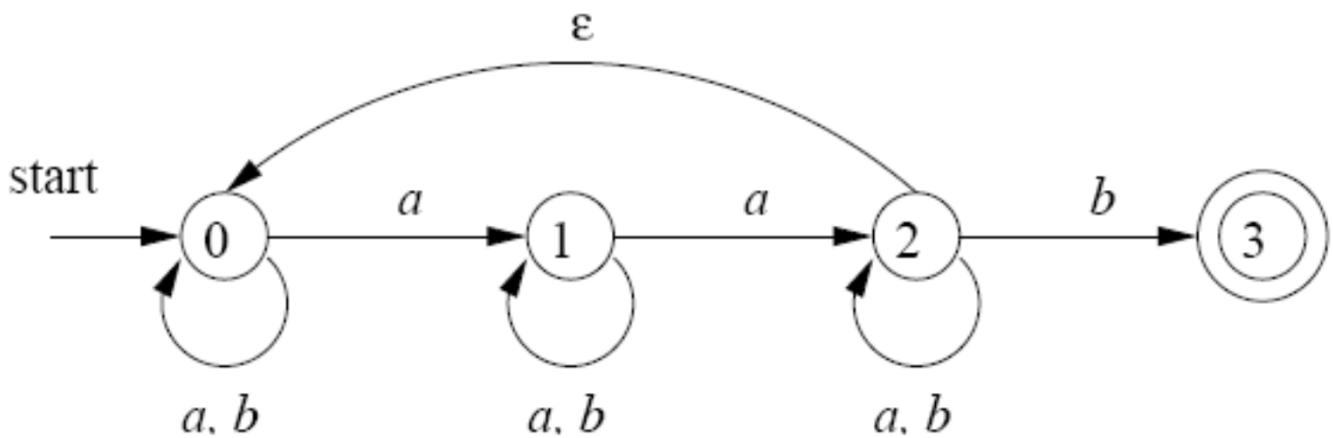
Crazy Semantics

https://www.destroyallsoftware.com/talks/wat

….Thank you Kevin Strasser

Things to Address…

- **Try these two at home**: Convert the NFA to DFA



  - Write a finite state automata that will recognize any string consisting of a and b characters where the number of a's is even (or zero) and the number of b's is odd.
- Test next Friday
- Milestone 2 due tonight and 3 will be posted this weekend.

*subset construction doesn't give a min DFA

Minimize a DFA

- First, group accepting states together and group other states together.

initially, let $\Pi_{\text{new}} = \Pi$;
for (each group G of Π) {
    partition G into subgroups such that two states s and t
        are in the same subgroup if and only if for all
        input symbols a, states s and t have transitions on a
        to states in the same group of Π;
    /* at worst, a state will be in a subgroup by itself */
    replace G in Π_{new} by the set of all subgroups formed;
}

Minimize a DFA for Lexical Analysis

Lex generators

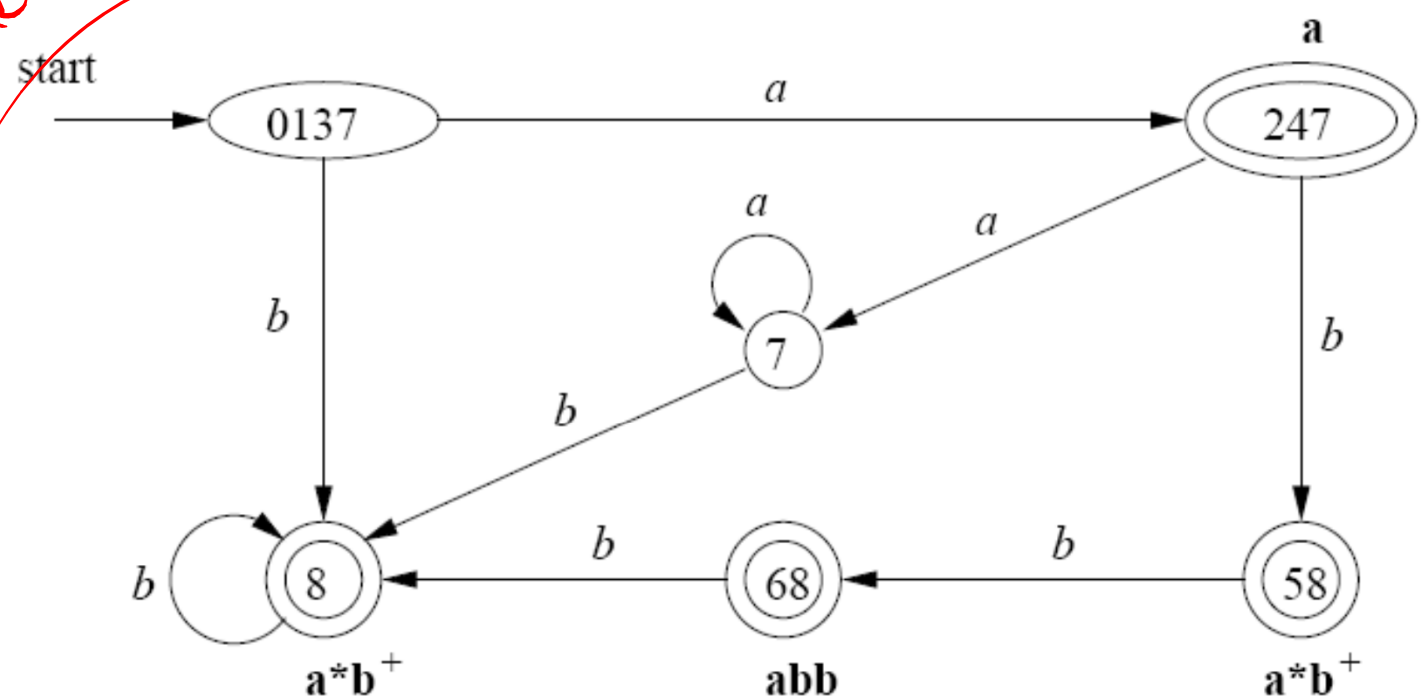


Figure 3.54: Transition graph for DFA handling the patterns **a**, **abb**, and $\mathbf{a}^*\mathbf{b}^+$

What is the Parser?

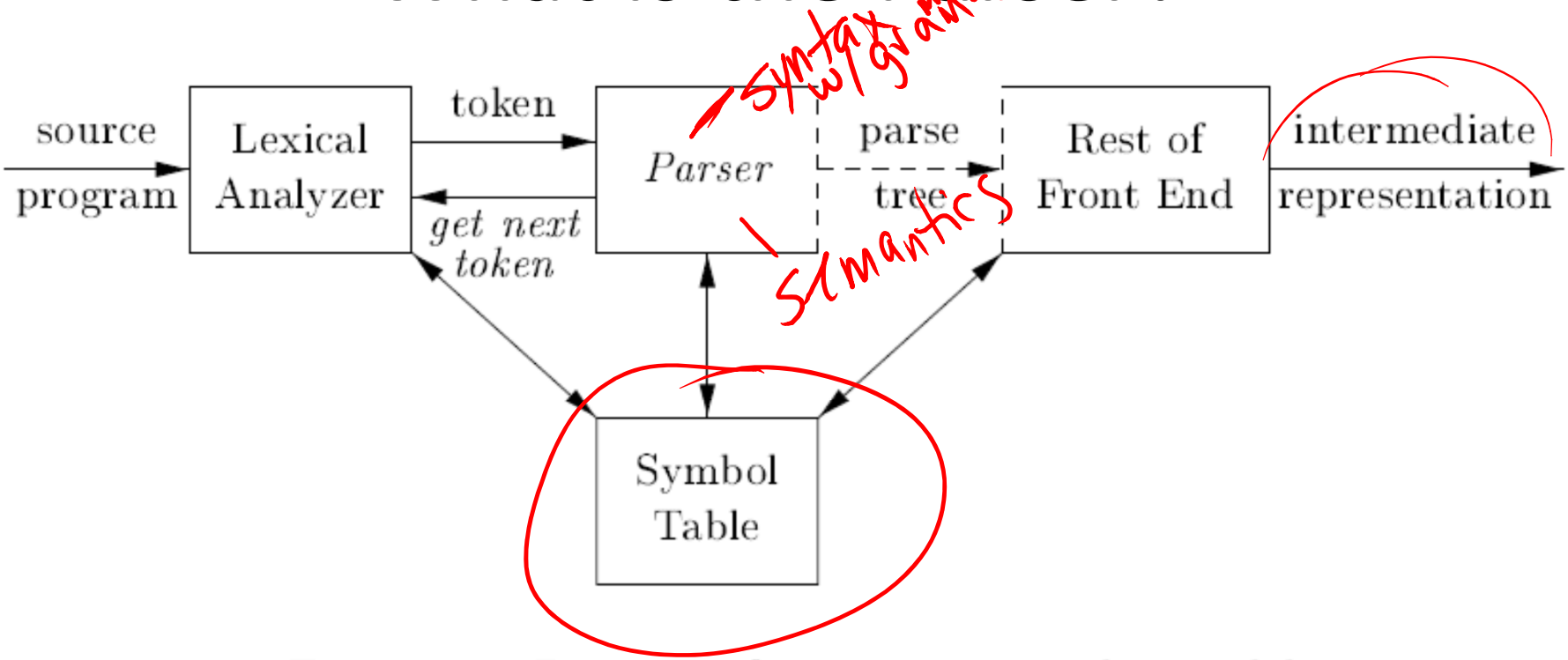


Figure 4.1: Position of parser in compiler model

Types of Parsers

- Universal
- Top-Down
- Bottom-Up

not used in practice - not efficient

any kind of grammar

recursive descent

simplest grammar LL

predictive parser

LR, SLR, LALR

CFG

Error Correction

changing the prefix

- Panic Mode — ditch tokens until you see a good one for the production
- Phrase-Level Recovery — inserting a "ex" terminal on the production when it gets bad token → missing semicolon & it inserts to cont.
- Error Productions — add productions to grammar of common mistakes
- ~~Global Recovery~~

Context Free Grammars

- Nonterminals, N
- Terminals, T
- Set of Productions, P
- Start Symbol, S

- Four-tuple (N, T, P, S)

Example

parsing

$$
\begin{array}{rcl}
\textit{expression} & \rightarrow & \textit{expression} + \textit{term} \\
\textit{expression} & \rightarrow & \textit{expression} - \textit{term} \\
\textit{expression} & \rightarrow & \textit{term} \\
\textit{term} & \rightarrow & \textit{term} * \textit{factor} \\
\textit{term} & \rightarrow & \textit{term} \ / \ \textit{factor} \\
\textit{term} & \rightarrow & \textit{factor} \\
\textit{factor} & \rightarrow & (\ \textit{expression}\ ) \\
\textit{factor} & \rightarrow & \textbf{id}
\end{array}
$$

why not a parsing - taken of by lex

Figure 4.2: Grammar for simple arithmetic expressions

Context Free vs. Regular Languages

- (a|b)*abb

  A -> aA | bA | abb

CFG

- L={$a^n b^n$ | n>=1}

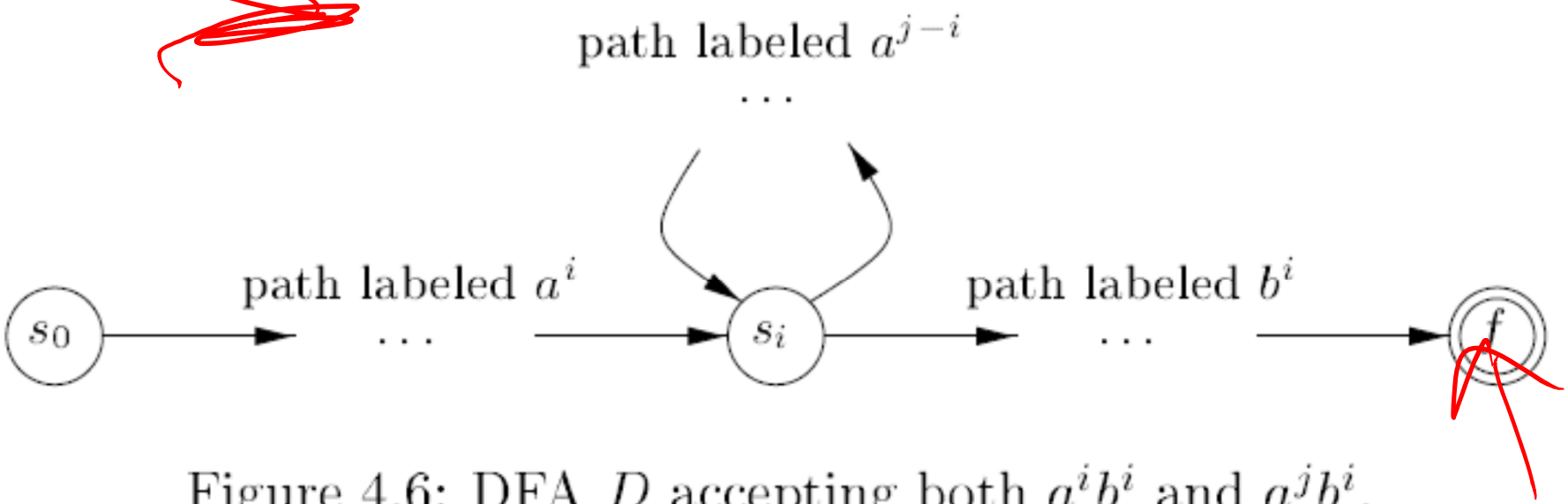


Figure 4.6: DFA D accepting both $a^i b^i$ and $a^j b^i$.

Production/Derivation Notation

- ::= vs. -> production
- $\Rightarrow$ ← derivation
- $\overset{*}{\Rightarrow}$ ← zero or more deri.
- $\overset{+}{\Rightarrow}$ ← one or more
- $\underset{lm}{\Rightarrow}$ ← leftmost or rightmost

$E \rightarrow E + E \mid E * E \mid (E) \mid \mathbf{id}$

$\cancel{(E) \rightarrow (id)}$

- Derivations
  - $E \Rightarrow (E)$
  - $E \Rightarrow (E) \Rightarrow (\mathbf{id})$
  - $E \overset{*}{\Rightarrow} (\mathbf{id})$ or $E \overset{+}{\Rightarrow} (\mathbf{id})$

only through one prod.

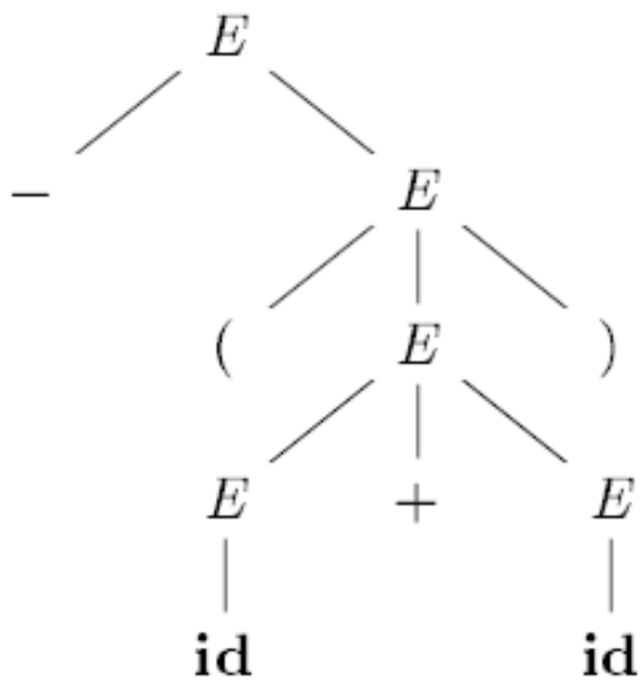
go through several prod.

recursive

$E \rightarrow E + E \mid E * E \mid - E \mid (E) \mid \mathbf{id}$

top down

- $E \underset{lm}{\Rightarrow} -E \underset{lm}{\Rightarrow} -(E + E) \underset{lm}{\Rightarrow} -(\mathbf{id} + E) \underset{lm}{\Rightarrow} -(\mathbf{id} + \mathbf{id})$
- $E \underset{rm}{\Rightarrow} -E \underset{rm}{\Rightarrow} -(E + E) \underset{rm}{\Rightarrow} -(E + \mathbf{id}) \underset{rm}{\Rightarrow} -(\mathbf{id} + \mathbf{id})$



bottom up

Ambiguity

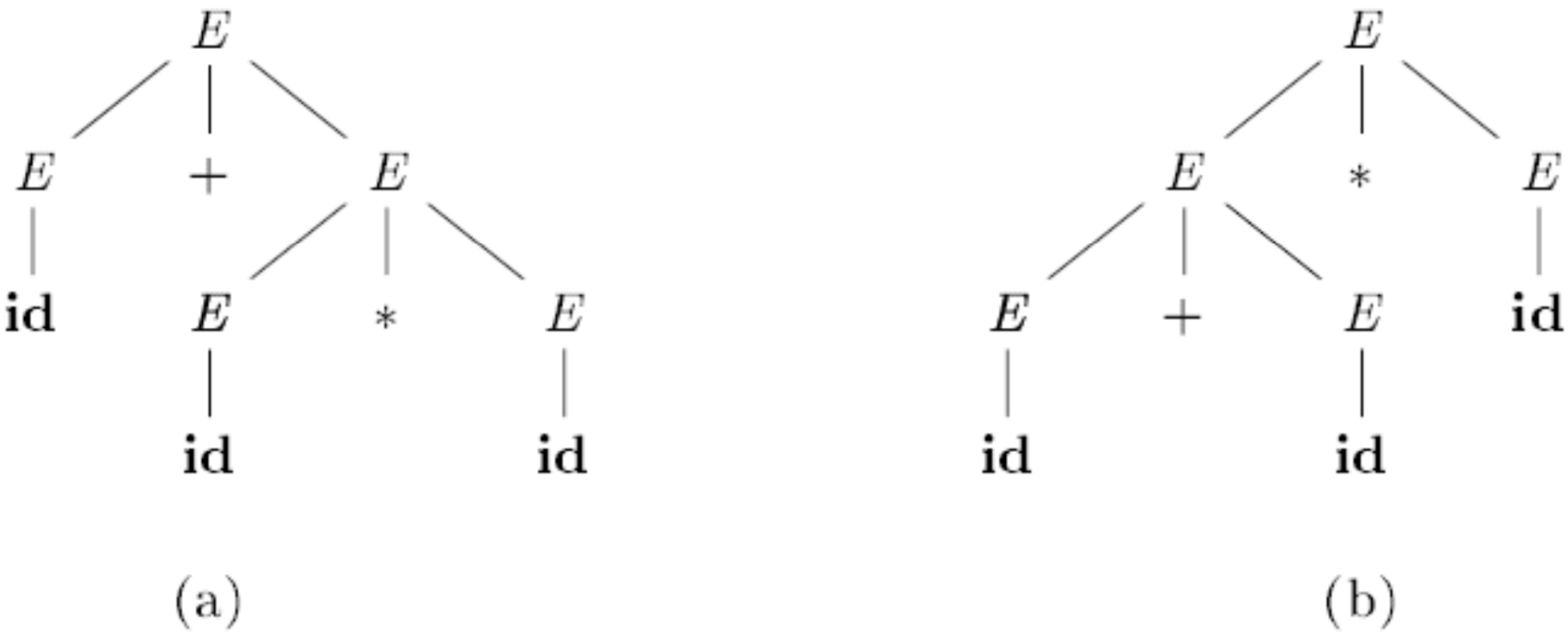


Figure 4.5: Two parse trees for **id+id*id**

derive in two ways w/ same type, i.e. LM or RM

Eliminate Left Recursion

recursive descent

- Immediate Left Recursion

  $A \to A\alpha_1 | A\alpha_2 | \ldots | A\alpha_m | \beta_1 | \beta_2 | \ldots | \beta_n$

  not

- $A \to \beta_1 A' | \beta_2 A' | \ldots | \beta_n A'$
- $A' \to \alpha_1 A' | \alpha_2 A' | \ldots | \alpha_m A' | \varepsilon$

- Example:

  $E \to E + E \mid E * E \mid (E) \mid \mathbf{id}$

  α_1 α_2 β_1 β_2

  $E \to (E)\, E' \mid \mathbf{id}\, E'$

  β_1 β_2

  $E' \to + E\, E' \mid * E\, E' \mid \varepsilon$

  α_1 α_2

Eliminate Left Recursion cont.

$A \to A\overset{\alpha_1}{c} \mid A\overset{\alpha_2}{a}d \mid \overset{\beta_1}{b}d \mid \overset{\beta_2}{\varepsilon}$

- Example:

  S -> A **a** | **b**

  A->A **c** | S **d** | ε

- Not immediate

  $S \Rightarrow A$ **a** $\Rightarrow S$ **d a**

$A \to \underline{bd}A' \mid A'$

$A' \to cA' \mid adA' \mid \varepsilon$

- Substitute all S productions in A

  A->A **c** | A **a d** | **b d** | ε

Eliminate Left Factoring

A-> $\alpha\beta_1 \mid \alpha\beta_2$

- A-> $\alpha A'$
- A' -> $\beta_1 \mid \beta_2$

- Example:

  stmt -> **if** *expr* **then** *stmt* **else** *stmt* | **if** *expr* **then** *stmt*

  α β_1 α β_2 ε

  stmt -> **if** *expr* **then** *stmt E*

  α

  E-> **else** *stmt* | ε

  β_1 β_2

make new non term for production

Top Down Parsing

```
void A() {
        Choose an A-production, A → X1 X2 ··· Xk;
        for ( i = 1 to k ) {
                if ( Xi is a nonterminal )
                        call procedure Xi();
                else if ( Xi equals the current input symbol a )
                        advance the input to the next symbol;
                else /* an error has occurred */;
        }
}
```

terms & non terms

A( )

- How does this change for the production below?

  A-> **ab** | **a**

a