# CSc 453 — Compilers and Systems Software

**6 : Top-Down Parsing I**

Christian Collberg
Department of Computer Science
University of Arizona
collberg@gmail.com

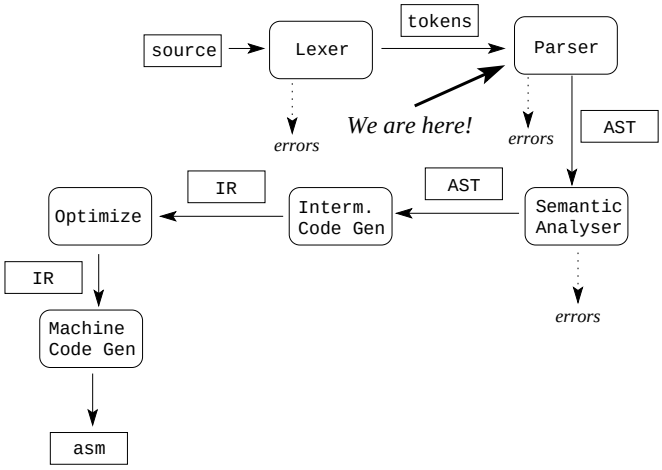
Copyright © 2009 Christian Collberg

September 14, 2009

## 1

# Overview

## 2 Compiler Phases




## 3

# Grammars

## 4 Context Free Grammars

- CFGs are used to describe the syntax of programming languages. A *production*

$$S \rightarrow \underline{\texttt{if}}\ E\ \underline{\texttt{then}}\ S_1\ \underline{\texttt{else}}\ S_2$$

in a CFG says

"If $S_1$ and $S_2$ are statements and $E$ an expression then '$\underline{\texttt{if}}$ $E$ $\underline{\texttt{then}}$ $S_1$ $\underline{\texttt{else}}$ $S_2$' is a statement".

Notice that this production is *recursive*; it allows if-statements to occur within if-statements.

# 5 Context Free Grammars...

$$S \rightarrow \underline{\texttt{if}}\ E\ \underline{\texttt{then}}\ S_1\ \underline{\texttt{else}}\ S_2$$

- $\underline{\texttt{if}}$, $\underline{\texttt{then}}$, and $\underline{\texttt{else}}$ are *terminal symbols* or *tokens*.
- $S$, $S_1$, $S_2$, and $E$ are *non-terminals*. They are like "variables", that represent the kinds of strings that the grammar defines as *statements* or *expressions*, respectively.

# 6 CFG Notation

**terminals:**
$\underline{\texttt{a}}, \underline{\texttt{b}}, \underline{\texttt{c}}, \cdots, \underline{\texttt{+}}, \underline{\texttt{-}}, \cdots, \underline{\texttt{0}}, \underline{\texttt{1}}, \cdots, \underline{\texttt{if}}, \underline{\texttt{do}}$.

**nonterminals:**
$\mathsf{A}, \mathsf{B}, \mathsf{C}, \cdots, \mathsf{S}, \cdots, \mathsf{expr}, \mathsf{stmt}$.

**grammar symbols:**
$X, Y, Z, \cdots$ (either terminals or nonterminals).

**strings of terminals:**
$u, v, w \cdots$.

**strings of grammar symbols:**
$\alpha, \beta, \gamma, \cdots$ (strings of terminals or nonterminals).

**productions:**
$A \rightarrow \alpha_1, A \rightarrow \alpha_2, \cdots, A \rightarrow \alpha_k$, or $A \rightarrow \alpha_1 \mid \alpha_2 \cdots \mid \alpha_k$.

# 7 Derivations — Productions as *Rewrite Rules*

1. Start with the *start symbol*, $S$.
2. Pick any production $S \rightarrow \alpha$, eg. $S \rightarrow \underline{\texttt{id}}\ \underline{\texttt{:=}}\ E$.
3. We say that $S$ *derives* $\underline{\texttt{id}}\ \underline{\texttt{:=}}\ E$, or $S \Rightarrow \underline{\texttt{id}}\ \underline{\texttt{:=}}\ E$. '$\underline{\texttt{id}}\ \underline{\texttt{:=}}\ E$' is a *sentential form* derived from $S$.
4. Repeat: pick a nonterminal $A$ from the sentential form, replace with the RHS of a production $A \rightarrow \alpha$: $S \Rightarrow \underline{\texttt{id}}\ \underline{\texttt{:=}}\ E \Rightarrow \underline{\texttt{id}}\ \underline{\texttt{:=}}\ E\underline{\texttt{+}}E \Rightarrow \underline{\texttt{id}}\ \underline{\texttt{:=}}\ \underline{\texttt{id}} + E \Rightarrow \underline{\texttt{id}}\ \underline{\texttt{:=}}\ \underline{\texttt{id+num}}$. $S \overset{*}{\Rightarrow} \underline{\texttt{id}}\ \underline{\texttt{:=}}\ \underline{\texttt{id+num}}$.

$$\begin{aligned} S &\rightarrow \underline{\texttt{id}}\ \underline{\texttt{:=}}\ E \mid \underline{\texttt{if}}\ E\ \underline{\texttt{then}}\ S \\ E &\rightarrow E\underline{\texttt{+}}E \mid \underline{\texttt{id}} \mid \underline{\texttt{num}} \end{aligned}$$

## 8 Terminology

- A grammar is a 4-tuple

$$\text{(non-terminals, terminals, productions, start-symbol)}$$

  or

$$(N, \Sigma, P, S)$$

- A production is of the form $\alpha \rightarrow \beta$ where $\alpha, \beta$ are taken from $N \bigcup \Sigma$.
- Read $\alpha \rightarrow \beta$ as "rewrite $\alpha$ with $\beta$".
- Read $\Rightarrow$ as "directly derives".
- Read $\overset{r}{\Rightarrow}$ as "directly derives using rule $r$".
- Read $\overset{*}{\Rightarrow}$ as "derives in zero or more steps".

## 9 Derivations...

- $\alpha A \beta \Rightarrow \alpha \gamma \beta$ if
  - $\boxed{A \rightarrow \gamma}$ is a production, and
  - $\alpha$ and $\beta$ are strings of grammar symbols.

**$\Rightarrow$:** Derives in one step.

**$\overset{*}{\Rightarrow}$:** Derives in 0 or more steps.

**$\overset{+}{\Rightarrow}$:** Derives in 1 or more steps.

**$\underset{lm}{\Rightarrow}$:** Leftmost derivation.

**$\underset{rm}{\Rightarrow}$:** Rightmost derivation.

**$L(G)$:** The language generated by grammar $G$. This is the set of strings $w$, such that there is a derivation $S \overset{+}{\Rightarrow} w$, where $S$ is $G$'s start-symbol.

## 10 Derivations...

The string of terminal symbols $\boxed{\underline{\texttt{id:=id+num}}}$ is generated by a leftmost derivation:

$$\begin{aligned} S &\underset{lm}{\Rightarrow} \underline{\texttt{id}}\ \underline{\texttt{:=}}\ E \underset{lm}{\Rightarrow} \underline{\texttt{id}}\ \underline{\texttt{:=}}\ E\underline{+}E \\ &\underset{lm}{\Rightarrow} \underline{\texttt{id}}\ \underline{\texttt{:=}}\ \underline{\texttt{id+}}E \underset{lm}{\Rightarrow} \underline{\texttt{id}}\ \underline{\texttt{:=}}\ \underline{\texttt{id+num}} \\ S &\overset{+}{\Rightarrow} \underline{\texttt{id:=id+num}} \end{aligned}$$

Example Grammar:

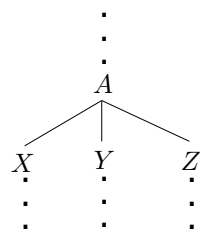
$$\begin{aligned} S &\rightarrow \underline{\texttt{id}}\ \underline{\texttt{:=}}\ E \mid \underline{\texttt{if}}\ E\ \underline{\texttt{then}}\ S \\ E &\rightarrow E\underline{+}E \mid \underline{\texttt{id}} \mid \underline{\texttt{num}} \end{aligned}$$

## 11 Parse Trees...

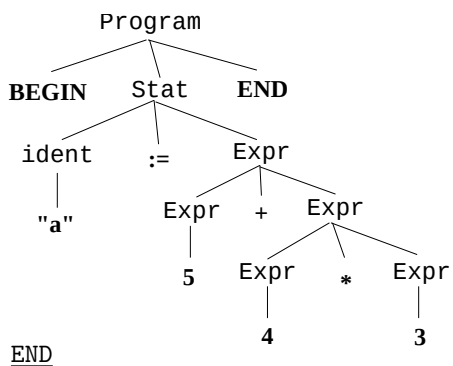
- If one step of our derivation is

$$\cdots A \cdots \Rightarrow \cdots X\ Y\ Z \cdots$$

(i.e, we used the rule $A \to XYZ$) then we'll get a parse (sub-)tree




## 12




Program $\Rightarrow$ BEGIN Stat END
$\Rightarrow$ BEGIN ident := Expr END
$\Rightarrow$ BEGIN "$a$" := Expr END
$\Rightarrow$ BEGIN "$a$" := Expr + Expr END
$\Rightarrow$ BEGIN "$a$" := 5 + Expr END
$\Rightarrow$ BEGIN "$a$" := 5 + Expr * Expr END
$\Rightarrow$ BEGIN "$a$" := 5 + 4 * Expr END
$\Rightarrow$ BEGIN "$a$" := 5 + 4 * 3 END

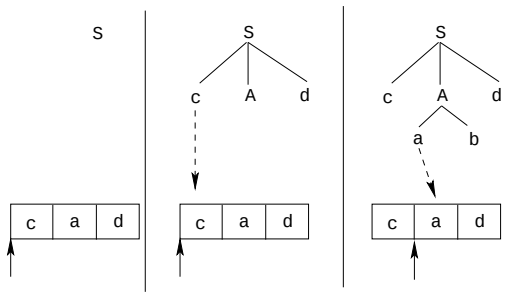
## 13

# Top-Down Parsing

## 14 Top-Down Backtracking Parser

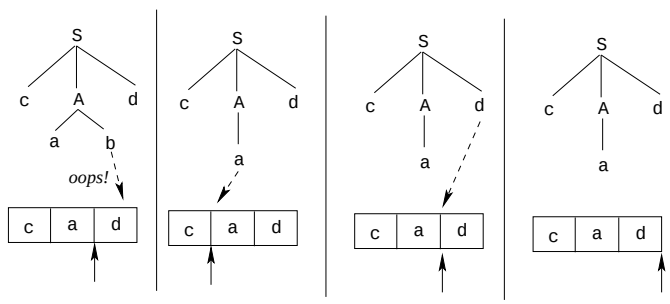
- Top-down parsing involves building a parse tree for the input string by starting at the root and adding nodes in preorder.

$$S \to cAd \qquad A \to ab|a$$




## 15 Top-Down Backtracking Parser...

- If a backtracking top-down parser chooses the wrong production rule to expand a node it backs up over the input, and undoes some of the parse tree construction:




## 16

# Grammar Rewriting

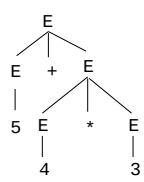
## 17 Ambiguous Grammars

- A grammar is ambiguous if some string of tokens can produce two (or more) different parse trees.

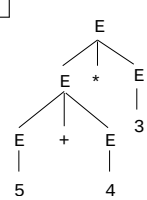
$$\mathtt{E} ::= \mathtt{E}\ \underline{+}\ \mathtt{E} \mid \mathtt{E}\ \underline{*}\ \mathtt{E} \mid \underline{\mathtt{number}}$$

$$\boxed{\mathtt{5\ +\ 4\ *\ 3}}$$

$$\begin{aligned}\mathtt{E} &\Rightarrow \mathtt{E}\ \underline{+}\ \mathtt{E}\\ &\Rightarrow \mathtt{5}\ \underline{+}\ \mathtt{E}\\ &\Rightarrow \mathtt{5}\ \underline{+}\ \mathtt{E}\ \underline{*}\ \mathtt{E}\\ &\Rightarrow \mathtt{5}\ \underline{+}\ \mathtt{4}\ \underline{*}\ \mathtt{E}\\ &\Rightarrow \mathtt{5}\ \underline{+}\ \mathtt{4}\ \underline{*}\ \mathtt{3}\end{aligned}$$




$$\begin{aligned}\mathtt{E} &\Rightarrow \mathtt{E}\ \underline{*}\ \mathtt{E}\\ &\Rightarrow \mathtt{E}\ \underline{*}\ \mathtt{3}\\ &\Rightarrow \mathtt{E}\ \underline{+}\ \mathtt{E}\ \underline{*}\ \mathtt{3}\\ &\Rightarrow \mathtt{E}\ \underline{+}\ \mathtt{4}\ \underline{*}\ \mathtt{3}\\ &\Rightarrow \mathtt{5}\ \underline{+}\ \mathtt{4}\ \underline{*}\ \mathtt{3}\end{aligned}$$




## 18 Operator Precedence

- The *precedence* of an operator is a measure of its *binding power*, i.e. how strongly it attracts its operands.

- Usually $*$ has higher precedence than $+$:
$$4 + 5 * 3$$
means
$$4 + (5 * 3),$$
not
$$(4 + 5) * 3.$$
- We say that $\underline{*}$ binds harder than $\underline{+}$.

## 19 Operator Associativity

- The *associativity* of an operator describes how operators of equal precedence are grouped.
- $+$ and $-$ are usually *left associative*:
$$4 - 2 + 3$$
means
$$(4 - 2) + 3 = 5,$$
not
$$4 - (2 + 3) = -1.$$
We say that $+$ *associates to the left.*
- ^ associates to the right:
$$2\hat{}3\hat{}4 = 2\hat{}(3\hat{}4).$$

## 20 Expression Grammars

- We must write unambiguous expression grammars that reflect the associativity and precedence of all operators.
- The next slide gives the algorithm for writing such grammars.

Resulting Expression Grammar:

expr ::= expr $\underline{+}$ term | term
term ::= term $\underline{*}$ factor | factor
factor ::= $\underline{(}$ expr $\underline{)}$ | $\underline{\texttt{number}}$

## 21

1. Create one non-terminal for each precedence level, for example $p_1, p_2, \cdots, p_n$, where $p_n$ has the highest precedence level.
2. For operator *op* at precedence level $i$ construct the following production if the operator is
   - left associative:

   $p_i$ ::= $p_i$ *op* $p_{i+1}$ | $p_{i+1}$

   - right associative:

   $p_i$ ::= $p_{i+1}$ *op* $p_i$ | $p_{i+1}$

3. Construct a production for nonterminal $p_{n+1}$ which represents *primary* expressions such as identifiers, numbers, parenthesized expressions, etc:

   $p_{n+1}$ ::= ($p_1$) | num | id

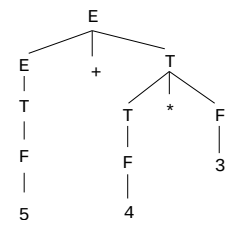
## 22

$$\mathsf{E} ::= \mathsf{E} \underline{+} \mathsf{T} \mid \mathsf{T}$$
$$\mathsf{T} ::= \mathsf{T} \underline{*} \mathsf{F} \mid \mathsf{F}$$
$$\mathsf{F} ::= \underline{\text{number}}$$

$$\boxed{5 + 4 * 3}$$

$$\begin{aligned}
\mathsf{E} &\Rightarrow \mathsf{E} \underline{+} \mathsf{T} \\
&\Rightarrow \mathsf{T} \underline{+} \mathsf{T} \\
&\Rightarrow \mathsf{F} \underline{+} \mathsf{T} \\
&\Rightarrow 5 \underline{+} \mathsf{T} \\
&\Rightarrow 5 \underline{+} \mathsf{T} \underline{*} \mathsf{F} \\
&\Rightarrow 5 \underline{+} \mathsf{F} \underline{*} \mathsf{F} \\
&\Rightarrow 5 \underline{+} 4 \underline{*} \mathsf{F} \\
&\Rightarrow 5 \underline{+} 4 \underline{*} 3
\end{aligned}$$

$$\begin{aligned}
\mathsf{E} &\Rightarrow \mathsf{E} \underline{+} \mathsf{T} \\
&\Rightarrow \mathsf{E} \underline{+} \mathsf{T} \underline{*} \mathsf{F} \\
&\Rightarrow \mathsf{E} \underline{+} \mathsf{T} \underline{*} 3 \\
&\Rightarrow \mathsf{E} \underline{+} \mathsf{F} \underline{*} 3 \\
&\Rightarrow \mathsf{E} \underline{+} 4 \underline{*} 3 \\
&\Rightarrow \mathsf{T} \underline{+} 4 \underline{*} 3 \\
&\Rightarrow \mathrm{F} \underline{+} 4 \underline{*} 3 \\
&\Rightarrow 5 \underline{+} 4 \underline{*} 3
\end{aligned}$$




## 23

# Top-Down Parsing

## 24 Recursive Descent Parsing

```
PROCEDURE S ();
   IF curr_tok = if THEN
      match(if); E();
      match(then); S();
   ELSIF curr_tok = id THEN
      match(id); match(:=); E();
   ELSE syntax error ENDIF;
PROCEDURE E ();
   IF curr_tok = id THEN match(id);
   ELSE IF curr_tok = num THEN match(num);
   ELSE E(); match(+); E();
   ENDIF;
```

$$\begin{aligned}
S &\rightarrow \underline{\texttt{id}}\ \underline{:=}\ E \\
&\mid\ \underline{\texttt{if}}\ E\ \underline{\texttt{then}}\ S \\
E &\rightarrow E \underline{+} E \\
&\mid\ \underline{\texttt{id}} \mid \underline{\texttt{num}}
\end{aligned}$$

## 25 Recursive Descent—*Small Problem 1*

We may loop forever:

```
PROCEDURE E ();
   IF ···
   ELSE E(); match(+); E();
   ···
```

## 26 Recursive Descent—*Small Problem 2*

What about productions that start out similarly:

$$\begin{array}{rcl} S & \rightarrow & \underline{\texttt{if}}\ E\ \underline{\texttt{then}}\ S\ | \\ & & \underline{\texttt{if}}\ E\ \underline{\texttt{then}}\ S\ \underline{\texttt{else}}\ S \end{array}$$

```
PROCEDURE S ();
  IF curr_tok = if THEN
    match(if); E(); match(then); S();
  ELSIF curr_tok = if THEN
    match(if); E(); match(then);
    S(); match(else); S();
  ELSIF ··· ENDIF
```

## 27 Recursive Descent—*Small Problem 3*

What if there are several possible "next" tokens:

$$\begin{array}{rcl} \text{prog} & \rightarrow & \text{decl}\ |\ \text{stat} \\ \text{stat} & \rightarrow & \underline{\texttt{if}}\cdots\ |\ \underline{\texttt{id()}}\ |\ \underline{\texttt{while}}\cdots \\ \text{decl} & \rightarrow & \underline{\texttt{int}}\ \underline{\texttt{id}}\ |\ \underline{\texttt{real}}\ \underline{\texttt{id}} \end{array}$$

```
PROCEDURE prog ();
   IF curr_tok ∈ {if, id, while} THEN stat();
   ELSIF curr_tok ∈ {int, real} THEN decl()
   ELSE syntax error ENDIF;
END;
PROCEDURE stat (); ··· END;
PROCEDURE decl (); ··· END;
```

## 28 Left Recursion Removal

*Left recursion* must be removed from the grammar, by turning it into *right recursion*: $A \rightarrow A\,\alpha \mid \beta \quad \Rightarrow \quad \begin{array}{rcl} A & \rightarrow & \beta \\ R & \rightarrow & \alpha \end{array}$

Example:

$$\begin{array}{rcl} \text{expr} & \rightarrow & \text{expr}\ \underline{+}\ \text{term}\ |\ \text{term} \\ & \Downarrow & \\ \text{expr} & \rightarrow & \text{term}\ R \\ R & \rightarrow & \underline{+}\ \text{term}\ R\ |\ \epsilon \end{array}$$

## 29 Left Recursion Removal...

- After left recursion removal, our expression grammar

$$\begin{array}{rcl} E & \rightarrow & E\underline{+}T|T \\ T & \rightarrow & T\underline{*}F|F \\ F & \rightarrow & \underline{(}\ E\ \underline{)}\ |\ \underline{\texttt{id}} \end{array}$$

turns into

$$
\begin{array}{rcl}
E & \rightarrow & T\ E' \\
E' & \rightarrow & \underline{\texttt{+}}\ T\ E' \mid \epsilon \\
T & \rightarrow & F\ T' \\
T' & \rightarrow & \underline{\texttt{*}}\ F\ T' \mid \epsilon \\
F & \rightarrow & \underline{\texttt{(}}\ E\ \underline{\texttt{)}} \mid \underline{\texttt{id}}
\end{array}
$$

## 30 Left Factoring

A top-down parser that reads input from left-to-right, can't choose between productions $E \rightarrow abF$ and $E \rightarrow abcF$. These must be *left factored*.

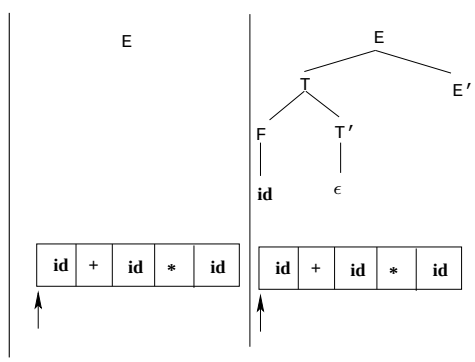
$$
\begin{array}{rcl}
A & \rightarrow & \alpha\ \beta_1 \ \mid\ \alpha\ \beta_2
\end{array}
\qquad \Rightarrow \qquad
\begin{array}{rcl}
A & \rightarrow & \alpha\ A' \\
A' & \rightarrow & \beta_1 \ \mid\ \beta_2
\end{array}
$$

Example:

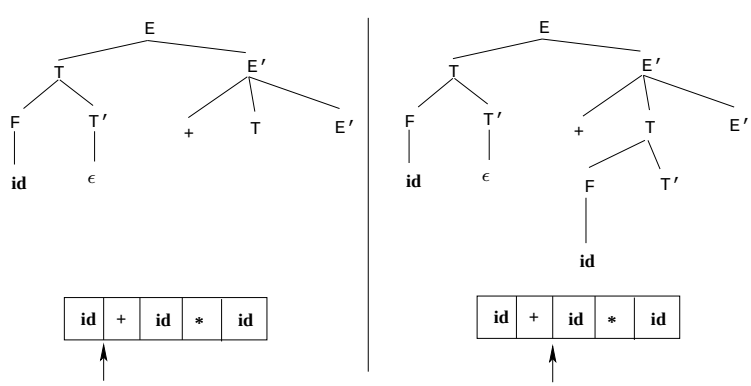
$$
\begin{array}{rcl}
S & \rightarrow & \underline{\texttt{if}}\ E\ \underline{\texttt{then}}\ S\ \underline{\texttt{else}}\ S \ \mid \\
 & & \underline{\texttt{if}}\ E\ \underline{\texttt{then}}\ S
\end{array}
\qquad \Rightarrow \qquad
\begin{array}{rcl}
S & \rightarrow & \underline{\texttt{if}}\ E\ \underline{\texttt{then}}\ S\ S' \\
S' & \rightarrow & \underline{\texttt{else}}\ S \ \mid\ \epsilon
\end{array}
$$

## 31 Top-Down Expression Parser

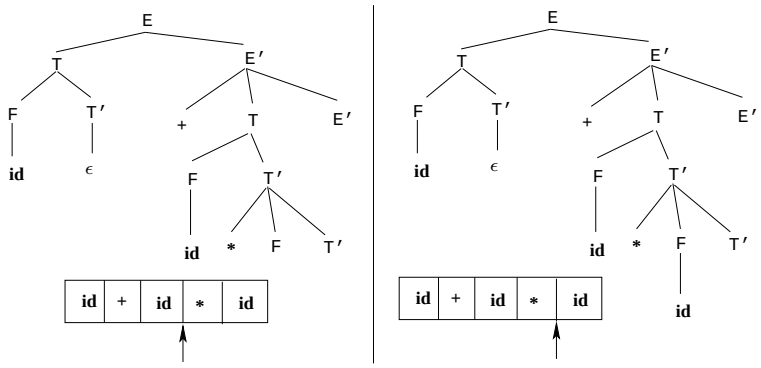
$$
\begin{array}{rcl}
E & \rightarrow & T\ E' \\
E' & \rightarrow & \underline{\texttt{+}}\ T\ E' \mid \epsilon \\
T & \rightarrow & F\ T' \\
T' & \rightarrow & \underline{\texttt{*}}\ F\ T' \mid \epsilon \\
F & \rightarrow & \underline{\texttt{(}}\ E\ \underline{\texttt{)}} \mid \underline{\texttt{id}}
\end{array}
$$



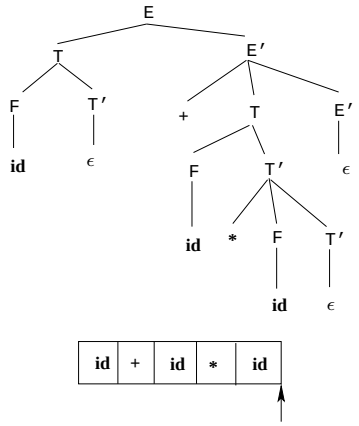

## 32 Top-Down Expression Parser...



## 33 Top-Down Expression Parser...




## 34 Top-Down Expression Parser...




## 35 Readings and References

- Read Louden, pp. 143–196.
- Or, the Dragon Book:

  **Top-Down Parsing** 181–190

  **Error Recovery** 192–195

  **Recursive Descent Parsing** 40–55, 75–76