CSc 453

Compilers and Systems Software

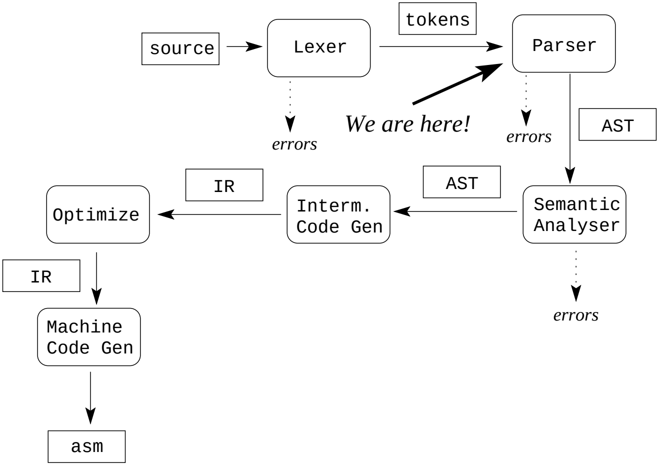
6 : Top-Down Parsing I

Department of Computer Science
University of Arizona

collberg@gmail.com

Overview

Compiler Phases



Grammars

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.

Context Free Grammars...

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

- $\underline{\texttt{if}}$, $\underline{\texttt{then}}$, 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.

CFG Notation

terminals:

$\underline{\mathtt{a}}, \underline{\mathtt{b}}, \underline{\mathtt{c}}, \cdots, \underline{\mathtt{+}}, \underline{\mathtt{-}}, \cdots, \underline{\mathtt{0}}, \underline{\mathtt{1}}, \cdots, \underline{\mathtt{if}}, \underline{\mathtt{do}}$.

nonterminals:

A, B, C, $\cdots$, S, $\cdots$, expr, 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:

$A \to \alpha_1, A \to \alpha_2, \cdots, A \to \alpha_k$, or

$A \to \alpha_1 \mid \alpha_2 \cdots \mid \alpha_k$.

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 $A \rightarrow \alpha$:
   $S \Rightarrow \underline{\texttt{id}}\ \underline{\texttt{:=}}\ E \Rightarrow \underline{\texttt{id}}\ \underline{\texttt{:=}}\ E\underline{\texttt{+}}E \Rightarrow$
   $\underline{\texttt{id}}\ \underline{\texttt{:=}}\ \underline{\texttt{id}} + E \Rightarrow \underline{\texttt{id}}\ \underline{\texttt{:=}}\ \underline{\texttt{id+num}}$. $S \stackrel{*}{\Rightarrow} \underline{\texttt{id}}\ \underline{\texttt{:=}}\ \underline{\texttt{id+num}}$.

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

Terminology

- A grammar is a 4-tuple

  (non-terminals, terminals, productions, start-symbol)

  or

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

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

Derivations...

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

$\Rightarrow$: Derives in one step.

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

$\stackrel{+}{\Rightarrow}$: 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 $S \stackrel{+}{\Rightarrow} w$, where S is G's start-symbol.

Derivations...

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

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

Example Grammar:

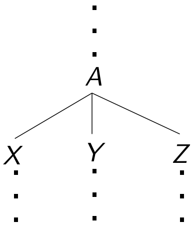
$$
\begin{array}{lll}
S & \rightarrow & \underline{\texttt{id}}\ \underline{\texttt{:=}}\ E \mid \underline{\texttt{if}}\ E\ \underline{\texttt{then}}\ S \\
E & \rightarrow & E\underline{\texttt{+}}E \mid \underline{\texttt{id}} \mid \underline{\texttt{num}}
\end{array}
$$

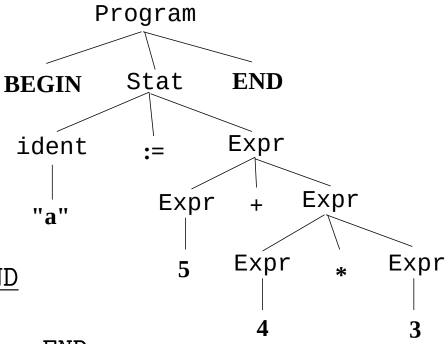
Parse Trees. . .

- 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





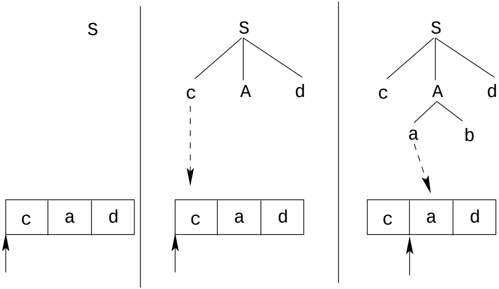
Program ⇒ BEGIN Stat END
⇒ BEGIN ident := Expr END
⇒ BEGIN "a" := Expr END
⇒ BEGIN "a" := Expr + Expr END
⇒ BEGIN "a" := 5 + Expr END
⇒ BEGIN "a" := 5 + Expr * Expr END
⇒ BEGIN "a" := 5 + 4 * Expr END
⇒ BEGIN "a" := 5 + 4 * 3 END

Top-Down Parsing

Top-Down Backtracking Parser

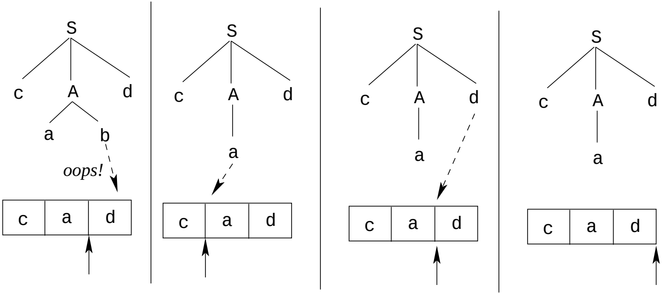
- Top-down parsing involves building a parse tree for the input string by starting at the root and adding nodes in preorder.

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



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:



Grammar Rewriting

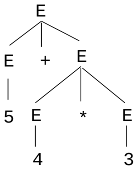
Ambiguous Grammars

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

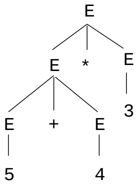
$$\mathrm{E} ::= \mathrm{E}\ \underline{+}\ \mathrm{E} \;—\; \mathrm{E}\ \underline{*}\ \mathrm{E} \;—\; \underline{\texttt{number}}$$

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

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



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



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 $\underline{+}$.

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.
- ˆ associates to the right:

  $$2\hat{}3\hat{}4 = 2\hat{}(3\hat{}4).$$

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 <u>+</u> term — term

term ::= term <u>*</u> factor — factor

factor ::= <u>(</u> expr <u>)</u> — <u>`number`</u>

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:

   $p_i ::= p_i \; op \; p_{i+1} \mid p_{i+1}$

   - right associative:

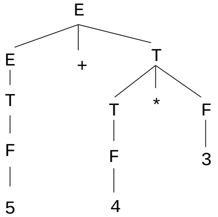
   $p_i ::= p_{i+1} \; op \; p_i \mid p_{i+1}$

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

   $p_{n+1} ::= (p_1) \mid \mathsf{num} \mid \mathsf{id}$

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

$$
\begin{aligned}
E &::= E \;\underline{+}\; T \;—\; T \\
T &::= T \;\underline{*}\; F \;—\; F \\
F &::= \underline{\texttt{number}}
\end{aligned}
$$



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

Top-Down Parsing

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);
   IF curr_tok = num THEN match(num);
   ELSE E(); match(+); E();
   ENDIF;
```

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

Recursive Descent—*Small Problem 1*

We may loop forever:

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

Recursive Descent—*Small Problem 2*

What about productions that start out similarly:

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

```
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
```

Recursive Descent—*Small Problem 3*

What if there are several possible "next" tokens:

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

```
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;
```

Left Recursion Removal

Left recursion must be removed from the grammar, by turning it into right recursion:

$$A \;\rightarrow\; A\,\alpha \mid \beta \quad\Rightarrow\quad \begin{array}{lll} A & \rightarrow & \beta\,R \\ R & \rightarrow & \alpha\,R \mid \epsilon \end{array}$$

Example:

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

Left Recursion Removal...

- After left recursion removal, our expression grammar

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

turns into

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

Left Factoring

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

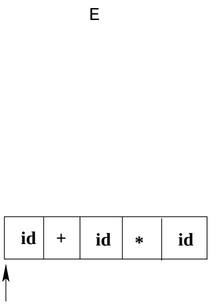
$$A \;\longrightarrow\; \alpha\,\beta_1 \mid \alpha\,\beta_2 \qquad \Rightarrow \qquad \begin{array}{rcl} A & \longrightarrow & \alpha\, A' \\ A' & \longrightarrow & \beta_1 \mid \beta_2 \end{array}$$

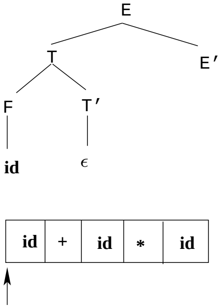
Example:

$$\begin{array}{rcl} S & \longrightarrow & \underline{\mathtt{if}}\ E\ \underline{\mathtt{then}}\ S\ \underline{\mathtt{else}}\ S \mid \\ & & \underline{\mathtt{if}}\ E\ \underline{\mathtt{then}}\ S \end{array} \quad \Rightarrow \quad \begin{array}{rcl} S & \longrightarrow & \underline{\mathtt{if}}\ E\ \underline{\mathtt{then}}\ S\ S' \\ S' & \longrightarrow & \underline{\mathtt{else}}\ S \mid \epsilon \end{array}$$

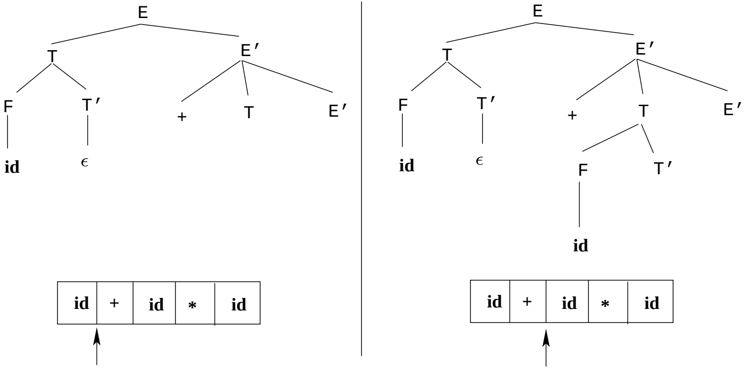
Top-Down Expression Parser

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

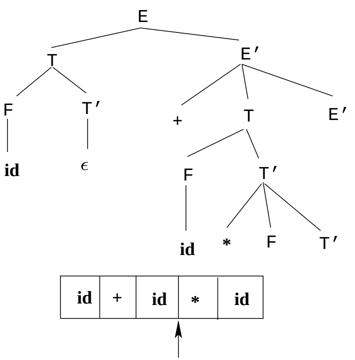


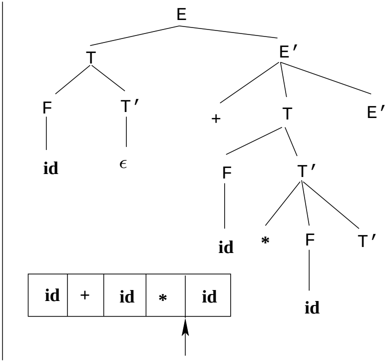


Top-Down Expression Parser. . .

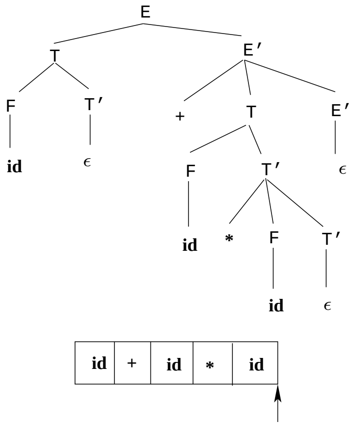


Top-Down Expression Parser. . .





Top-Down Expression Parser. . .



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