

CS321 Languages and Compiler Design I

Fall 2010

Lecture 7

PARSING

A **parser** is a program that, given an input sentence, “recognizes” whether or not that sentence is in the language of a given grammar.

Works by reconstructing a **derivation** for the sentence.

Parser “constructs” parse tree:

- **explicitly** – actual data structure is built; or
- **implicitly** – “semantic actions” are invoked at points corresponding to nodes in the tree, but no tree is actually built.

All parsers read input **left-to-right**, but they differ in how tree is constructed: **top-down** vs. **bottom-up**.

Any context-free grammar can be parsed by a (nondeterministic) pushdown automaton (NFA + stack), but not necessarily by a deterministic one (much less an efficient one).

TOP-DOWN PARSING

Idea: construct parse tree by starting at start symbol and “guessing” each derivation until we reach a string that matches input.

Example Grammar:

$$\begin{aligned} S &\rightarrow \text{if } E \text{ then } S \text{ else } S \mid \text{while } E \text{ do } S \mid \text{print} \mid \epsilon \\ E &\rightarrow \text{true} \mid \text{false} \mid \text{id} \end{aligned}$$

Token string:

if id_b then while true do else print

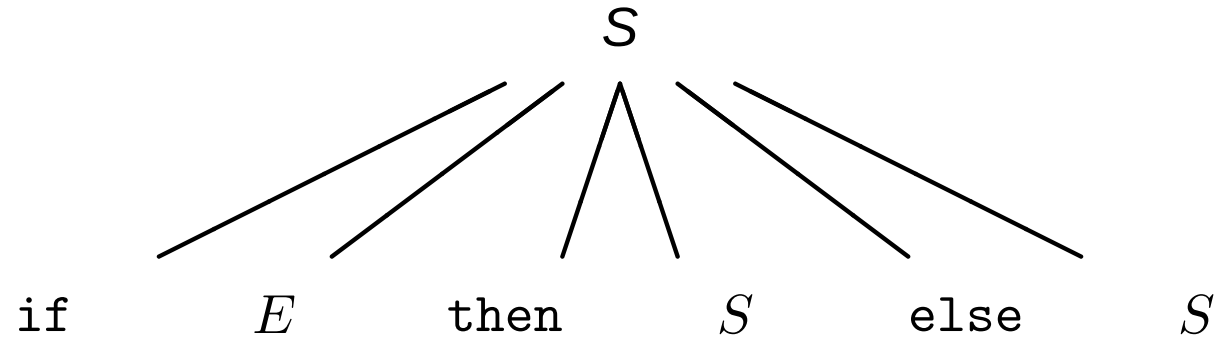
Tree:

S

Input: if id_b then while true do else print

Action: Guess for S

Tree:

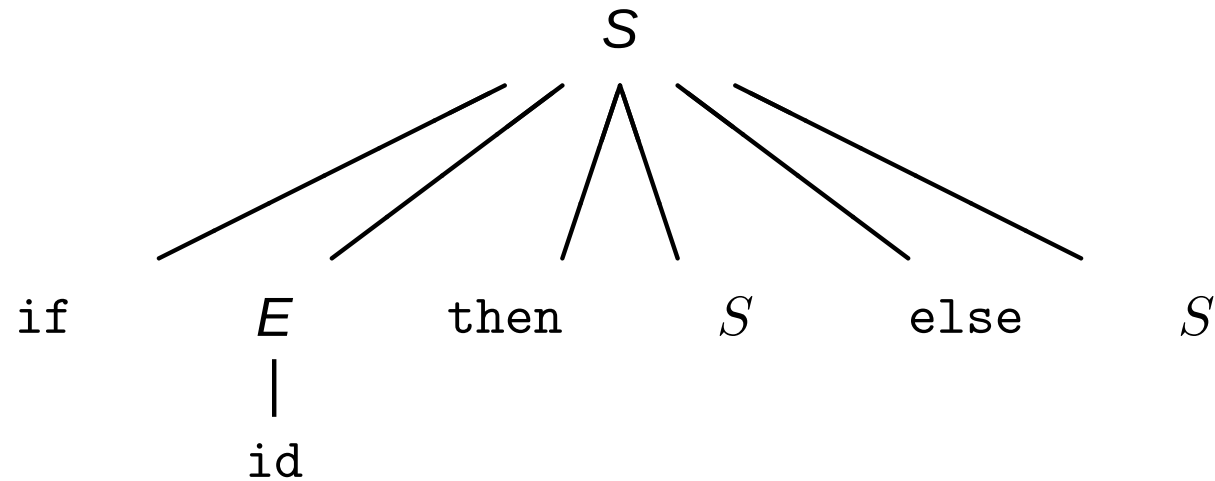


Input: if id_b then while true do else print

Action:

if matches; guess for E

Tree:

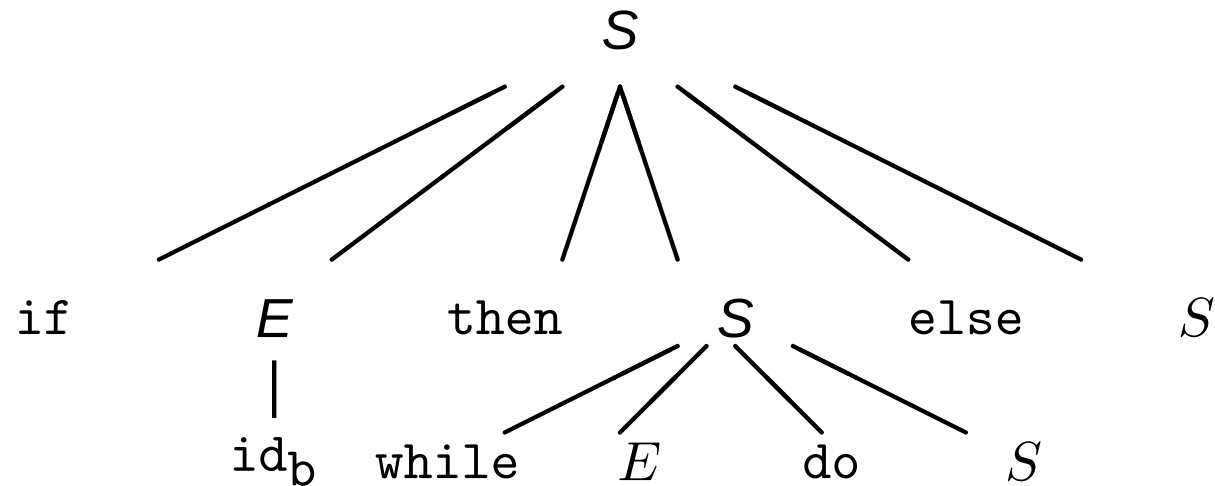


Input: id_b then while true do else print

Action:

id_b matches; then matches; guess for S

Tree:

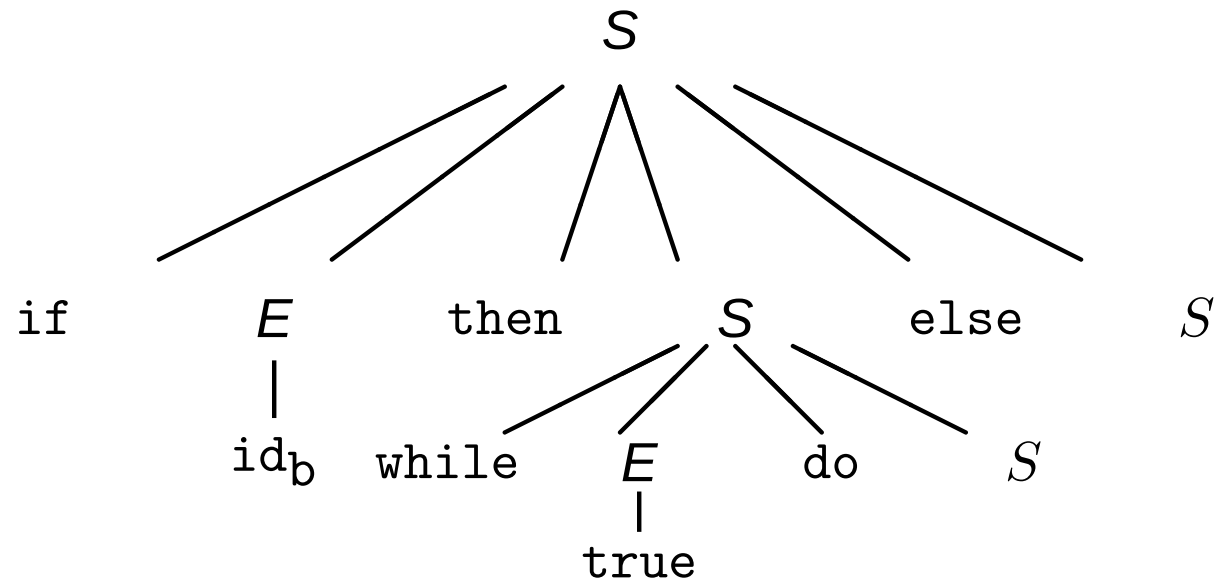


Input: while true do else print

Action:

while matches; guess for E

Tree:

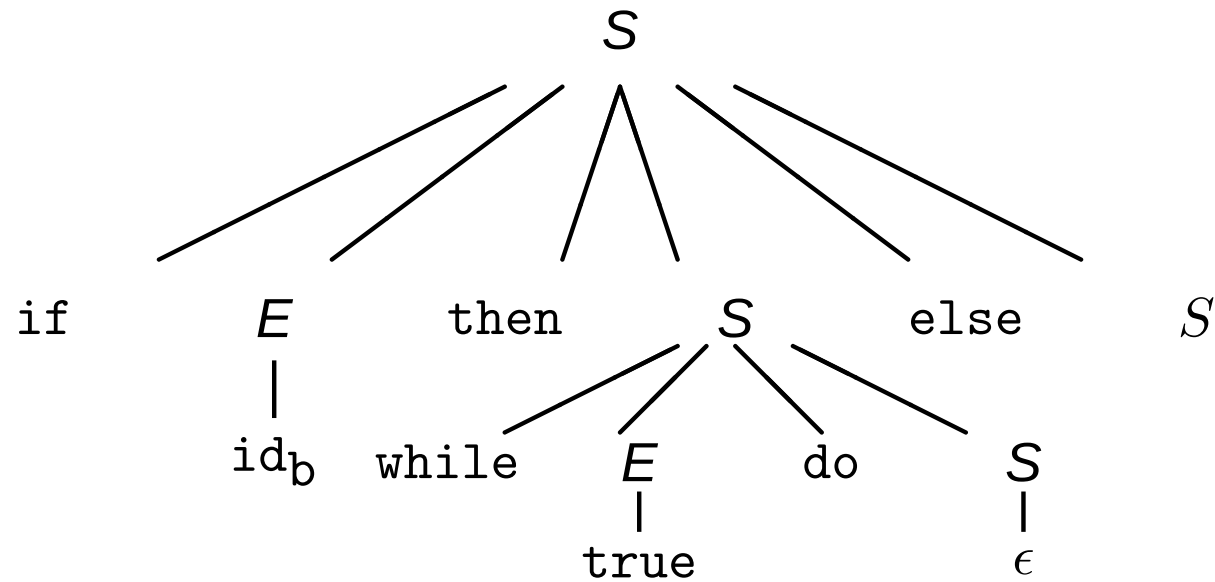


Input: true do else print

Action:

true matches; do matches; guess for S

Tree:

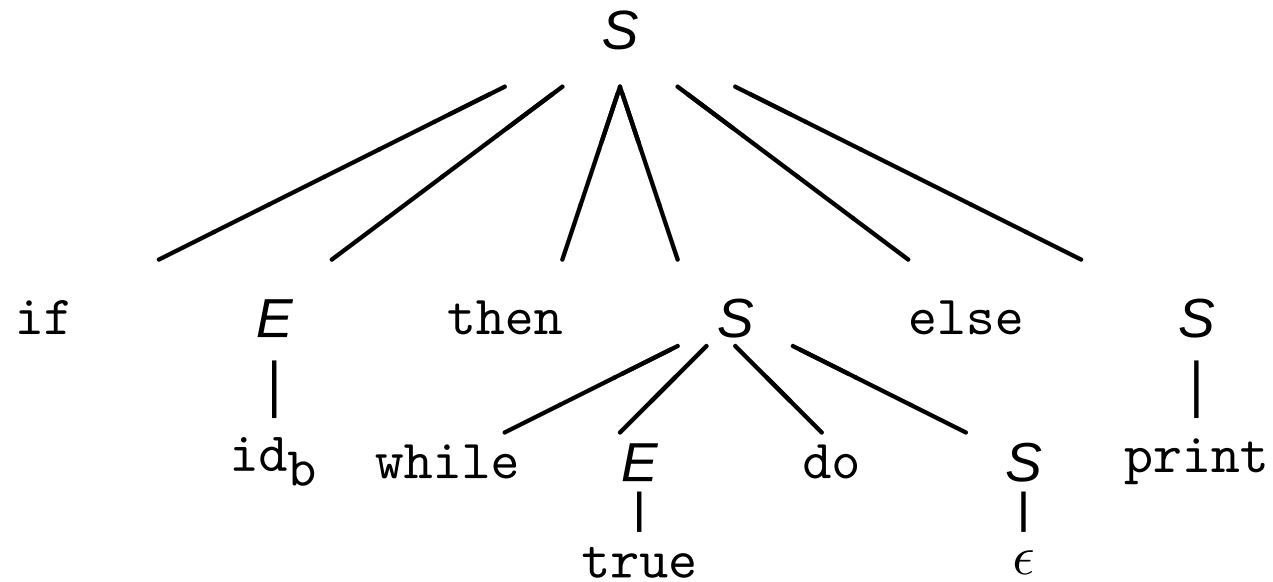


Input: else print

Action:

ϵ matches; else matches; guess for S

Tree:



Input: print

Action:

print matches; input exhausted; done.

RECURSIVE-DESCENT PARSING

Implementation of top-down parser using a **recursive** procedure for each non-terminal.

For many languages, can make perfect guesses (avoid back-tracking) by using 1-symbol **lookahead**. I.e., if

$$A \rightarrow \alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_n$$

choose correct α_i by looking at **first** symbol it derives.

(If ϵ is an alternative, choose it last.)

This approach is also called **predictive parsing**

Recursive-descent parsers are easy to write by hand and reasonably efficient.

Often must massage grammar into suitable form (more later).

Not all languages can be parsed this way.

RECURSIVE-DESCENT EXAMPLE

For the same grammar as before:

$$\begin{aligned} S &\rightarrow \text{if } E \text{ then } S \text{ else } S \mid \text{while } E \text{ do } S \mid \text{print } \epsilon \\ E &\rightarrow \text{true} \mid \text{false} \mid \text{id} \end{aligned}$$

We write one function to parse expressions, and another to parse statements.

Each function returns normally if it successfully parsed; otherwise it calls `error()`, which we assume issues an error message and terminates the parser.

For simplicity in this example, the parser functions don't return anything.

We assume that `lex()` returns the next token and advances the lexical analyzer's token stream.

```
void e() {  
    if (tok == TRUE || tok == FALSE || tok == ID)  
        tok = lex();  
    else error();  
}
```

```

void s() {
    if (tok == IF) {
        tok = lex(); /* get next input token */
        e();
        if (tok == THEN) {
            tok = lex();
            s(); /* recursive call! */
            if (tok == ELSE) {
                tok = lex();
                s();
            } else error(); /* issue error message */
        } else error();
    } else if (tok == WHILE) {
        tok = lex();
        e();
        if (tok == DO) {
            tok = lex();
            s();
        } else error();
    } else if (tok == PRINT) {
        tok = lex();
    } else
        /* epsilon case falls out */
}

```

PROBLEMS FOR RECURSIVE-DESCENT PARSING

- Left recursion: a derivation

$$A \xRightarrow{*} A\alpha$$

causes parser to loop!

Solution: **Remove** left recursion from grammar.

- Need to backtrack (inefficient) because one-symbol lookahead can't "guess" correctly, e.g.:

$$S \rightarrow V := \text{int}$$

$$V \rightarrow \text{alpha } '[' \text{ int } ']' \mid \text{alpha}$$

Possible inputs: $x := 77$ or $x[2] := 17$.

Which alternative should we choose for V ?

Solution: **Left-factor** the grammar.

- These problems arise naturally in expression grammars. (Can usually prevent them in statement grammars by careful language design.)

ELIMINATING IMMEDIATE LEFT RECURSION

Replace left-recursive productions of the form

$$A \rightarrow A\alpha \mid \beta$$

which generate sentences of the form

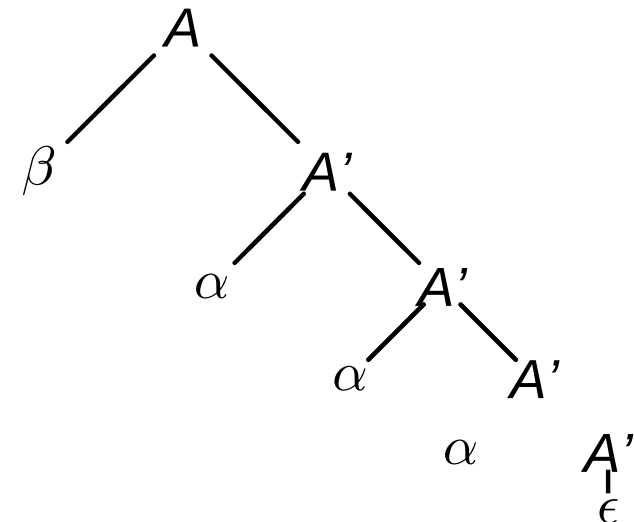
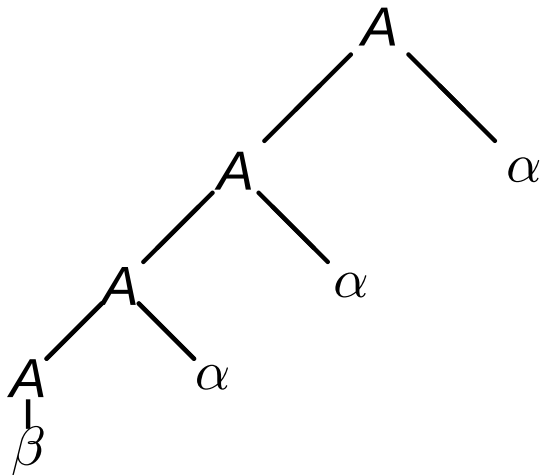
$$\beta, \beta\alpha, \beta\alpha\alpha, \dots$$

by the **right-recursive** productions

$$A \rightarrow \beta A'$$

$$A' \rightarrow \alpha A' \mid \epsilon$$

Yields different parse trees but same language:



Left-Recursion Removal in Arithmetic Expressions

For a simplified expression grammar:

$$E \rightarrow E + T \mid E - T \mid T$$

$$T \rightarrow T * F \mid T / F \mid F$$

$$F \rightarrow (E) \mid \text{id}$$

becomes

$$E \rightarrow T E'$$

$$E' \rightarrow + T E' \mid - T E' \mid \epsilon$$

$$T \rightarrow F T'$$

$$T' \rightarrow * F T' \mid / F T' \mid \epsilon$$

$$F \rightarrow (E) \mid \text{id}$$

But note that the desired left-associativity has been lost!

ELIMINATING ALL LEFT-RECURSION

Consider
$$\begin{array}{lcl} S & \rightarrow & Aa \mid b \\ A & \rightarrow & Sc \mid d \end{array}$$

Non-terminal S is left-recursive in two steps:

$S \Rightarrow Aa \Rightarrow Sca \Rightarrow Aaca \Rightarrow Scaca \Rightarrow \dots$

Fairly General Algorithm

(Works unless $A \xRightarrow{+} A$ or $A \rightarrow \epsilon$. Fully general algorithm exists, but is complicated!)

- arrange non-terminals in some order $A_1, \dots, A_n$.
- for $i = 1$ to n do
 - for $j = 1$ to $i-1$ do
 - for any production $A_i \rightarrow A_j \alpha$
replace it by substituting definition of A_j into r.h.s.,
i.e., by changing it to
$$A_i \rightarrow \beta_1 \alpha \mid \dots \mid \beta_m \alpha$$
where current productions for A_j are
$$A_j \rightarrow \beta_1 \mid \dots \mid \beta_m$$
- eliminate any immediate left-recursion in A_i

Example: Consider a grammar with the non-terminals ordered as follows:

$$(1) \quad S \rightarrow Pa \mid b$$

$$(2) \quad P \rightarrow Sc \mid Rd$$

$$(3) \quad R \rightarrow Se \mid f$$

Step i=2,j=1: In-line (1) into (2), giving

$$(2) \quad P \rightarrow Pac \mid bc \mid Rd$$

and then remove immediate left-recursion, giving

$$(2) \quad P \rightarrow bcP' \mid RdP'$$

$$(2') \quad P' \rightarrow acP' \mid \epsilon$$

Step i=3,j=1: Inline (1) into (3), giving

$$(3) \quad R \rightarrow Pae \mid be \mid f$$

Step i=3,j=2: Inline new (2) into new (3), giving

$$(3) \quad R \rightarrow bcP'ae \mid RdP'ae \mid be \mid f$$

and then remove immediate left-recursion, giving

$$(3) \quad R \rightarrow bcP'aeR' \mid beR' \mid fR'$$

$$(3') \quad R' \rightarrow dP'aeR' \mid \epsilon$$

LEFT-FACTORING

Easy to remove common prefixes by left-factoring, creating new non-terminals.

Change

$$V \rightarrow \alpha\beta \mid \alpha\gamma$$

to

$$V \rightarrow \alpha V'$$

$$V' \rightarrow \beta \mid \gamma$$

Example:

$$S \rightarrow V := \text{int}$$

$$V \rightarrow \text{alpha } '[' \text{ int } ']' \mid \text{alpha}$$

becomes

$$S \rightarrow V := \text{int}$$

$$V \rightarrow \text{alpha } V'$$

$$V' \rightarrow '[' \text{ int } ']' \mid \epsilon$$

EXPRESSION PARSING USING RECURSIVE DESCENT (I)

Recall grammar of arithmetic expressions after removal of left recursion:

$$E \rightarrow TE'$$

$$E' \rightarrow + TE' \mid \epsilon$$

$$T \rightarrow FT'$$

$$T' \rightarrow * FT' \mid \epsilon$$

$$F \rightarrow (E) \mid \text{id}$$

This leads directly to this code for E and E' (T and T' are similar):

```
e() {  
    t();  
    e1();  
}  
  
e1() {  
    if (tok == '+') {  
        tok = lex(); t(); e1();  
    }  
    /* epsilon case falls through */  
}
```

EXPRESSION PARSING USING RECURSIVE-DESCENT (II)

We can simplify this code (and improve its performance) by turning the recursions into iterations, e.g.:

```
e1() {  
    if (tok == '+') {  
        tok = lex(); t(); e1();  
    }  
    /* epsilon case falls through */  
}
```

becomes:

```
e1() {  
    while (tok == '+') {  
        tok = lex(); t();  
    }  
}
```

EXPRESSION PARSING USING RECURSIVE-DESCENT (III)

We then inline functions that are (now) called only once, e.g.:

```
e() { t(); e1(); }
```

becomes:

```
e() {  
    t();  
    while (tok == '+') {  
        tok = lex(); t();  
    }  
}
```

Performing the same transformation on `t` and `t1`, and adding the usual recursive descent code for `f`, we get...

EXPRESSION PARSING USING RECURSIVE-DESCENT (IV)

```
e() {  
    t();  
    while (tok == '+') { lex(); t(); }  
}
```

```
t() {  
    f();  
    while (tok == '*') { lex(); f(); }  
}
```

```
f() {  
    if (tok == ID)  
        lex();  
    else if (tok == '(') {  
        lex();  
        e();  
        if (tok == ')')  
            lex();  
        else error();  
    }  
    else error();  
}
```

