

CS321 F'04 Lecture Notes
Lecture 7

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:

$S \rightarrow \text{if } E \text{ then } S \text{ else } S \mid \text{while } E \text{ do } S \mid \text{print}$
 $E \rightarrow \text{true} \mid \text{false} \mid \text{id}$

Token string:

if id then while true do print else print

Tree:

S

Input:

if id then while true do print else print

Action: Guess for S

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.

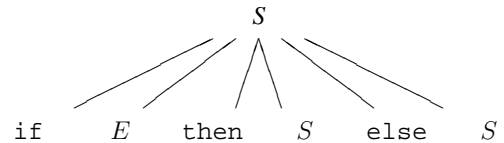
Parser “constructs” parse tree:

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

Tree:



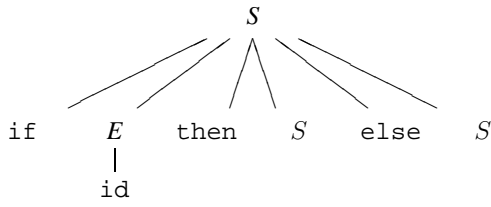
Input:

if id then while true do print else print

Action:

if matches; guess for E

Tree:



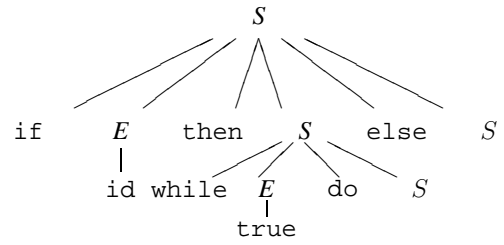
Input:

id then while true do print else print

Action:

id matches; then matches; guess for *S*

Tree:

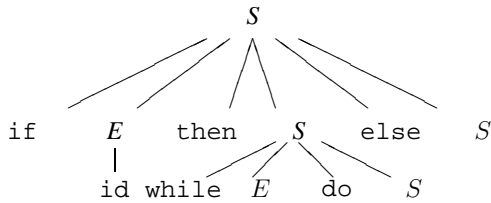


Input: true do print else print

Action:

true matches; do matches; guess for *S*

Tree:

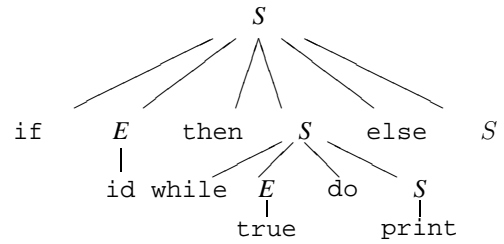


Input: while true do print else print

Action:

while matches; guess for *E*

Tree:

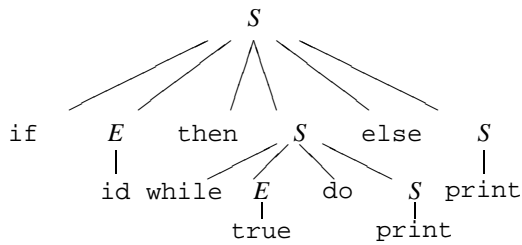


Input: print else print

Action:

print matches; else matches; guess for *S*

Tree:



Input: print

Action:

print matches; input exhausted; done.

Recursive-Descent Example

$S \rightarrow \text{if } E \text{ then } S \text{ else } S \mid \text{while } E \text{ do } S \mid \epsilon$
 $E \rightarrow \text{true} \mid \text{false} \mid \text{id}$

```

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();
  }
  /* epsilon case falls out */
}

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

Recursive-Descent Parsing

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

For many languages, can make perfect guesses (avoid backtracking) 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**

R.D. 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.

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.:

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

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

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

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

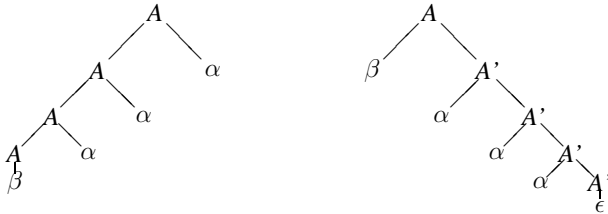
which generate sentences of the form

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

by the **right-recursive** productions

$$\begin{aligned} A &\rightarrow \beta A' \\ A' &\rightarrow \alpha A' \mid \epsilon \end{aligned}$$

Yields different parse trees but same language:



Example: Arithmetic Expressions

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

becomes

$$\begin{aligned} T &\rightarrow F T' \\ T' &\rightarrow * F T' \mid / F T' \mid \epsilon \end{aligned}$$

Left-factoring

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

Change

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

to

$$\begin{aligned} V &\rightarrow \alpha V' \\ V' &\rightarrow \beta \mid \gamma \end{aligned}$$

Example:

$$\begin{aligned} S &\rightarrow V := \text{int} \\ V &\rightarrow \text{alpha '[' int ']' } \mid \text{alpha} \end{aligned}$$

becomes

$$\begin{aligned} S &\rightarrow V := \text{int} \\ V &\rightarrow \text{alpha } V' \\ V' &\rightarrow \text{'[' int ']' } \mid \epsilon \end{aligned}$$

Eliminating all left-recursion

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

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 $A \xrightarrow{\pm} A$ 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 defn. of A_j into r.h.s., i.e., by $A_i \rightarrow \beta_1\alpha \mid \dots \mid \beta_m\alpha$ 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: Replace $A \rightarrow Sc \mid d$

by $A \rightarrow Aac \mid bc \mid d$

and then by $\begin{aligned} A &\rightarrow bcA' \mid dA' \\ A' &\rightarrow acA' \mid \epsilon \end{aligned}$

Expression Parsing Using Recursive Descent(I)

Consider a simple grammar of arithmetic expressions, with precedence expressed by multiple levels of non-terminals:

$$\begin{aligned} E &\rightarrow E + T \mid T \\ T &\rightarrow T * F \mid F \\ F &\rightarrow (E) \mid \text{id} \end{aligned}$$

To prepare it for recursive-descent parsing, we first remove left-recursion:

$$\begin{aligned} E &\rightarrow TE' \\ E' &\rightarrow + TE' \mid \epsilon \\ T &\rightarrow FT' \\ T' &\rightarrow * FT' \mid \epsilon \\ F &\rightarrow (E) \mid \text{id} \end{aligned}$$

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();
  }
}
```

and then inlining functions now called only once, e.g.:

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

becomes:

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

Expression parsing using recursive-descent (III)

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

```
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();
}
```