CS301 W'99 Lecture Notes
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 CFG 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:

$S \rightarrow$ `if` E `then` S `else` S | `while` E `do` S | `pause`
$E \rightarrow$ `true` | `false` | `id`

Token string:

`if id then while true do pause else pause`

Tree:

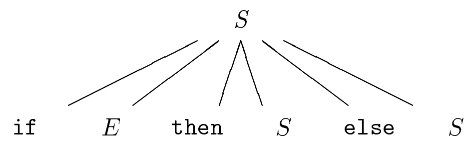
S

Input:

`if id then while true do pause else pause`

Action: Guess for S

Tree:



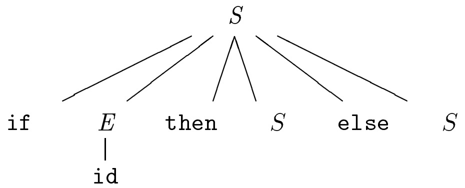
Input:

`if id then while true do pause else pause`

Action:

`if` matches; guess for E

Tree:



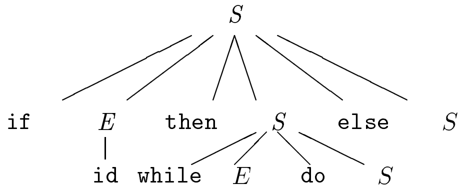
Input:

`id then while true do pause else pause`

Action:

`id` matches; `then` matches; guess for S

Tree:

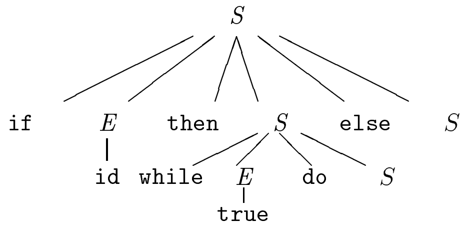


Input: `while true do pause else pause`

Action:

`while` matches; guess for E

Tree:

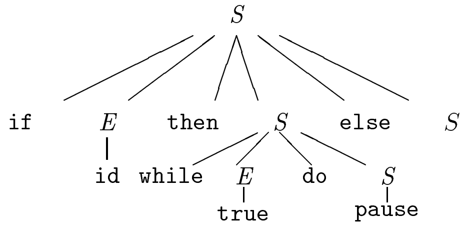


Input: `true do pause else pause`

Action:

`true` matches; `do` matches; guess for S

Tree:

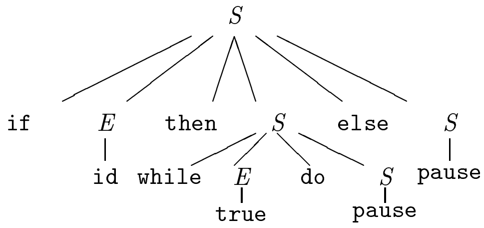


Input: `pause else pause`

Action:

`pause` matches; `else` matches; guess for S

Tree:



Input: `pause`

Action:

`pause` 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 backtracking) by using 1-symbol **lookahead**. I.e., if

$A \to \alpha_1 | \alpha_2 | \ldots | \alpha_n$

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.

Recursive-Descent Example

$S \to$ `if` E `then` S `else` S | `while` E `do` S | ϵ
$E \to$ `true` | `false` | `id`

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

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


Problems for Recursive-Descent Parsing

• Left recursion: a derivation

$A \stackrel{*}{\Rightarrow} A\alpha$

causes parser to loop!

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

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

$S \to V$ `:= int`
$V \to$ `alpha '[' int ']'` | `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

$A \rightarrow A\alpha | \ \beta$

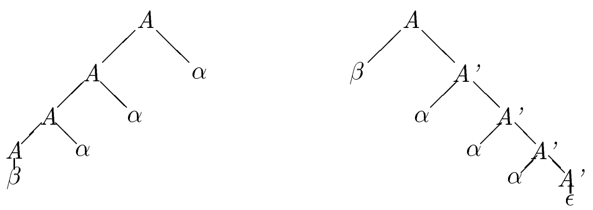
which generate sentences of the form

$\beta, \ \beta\alpha, \ \beta\alpha\alpha, \ \ldots$

by the **right-recursive** productions

$$\begin{array}{lcl} A & \rightarrow & \beta A' \\ A' & \rightarrow & \alpha A' | \ \epsilon \end{array}$$

Yields different parse trees but same language:



Example: Arithmetic Expressions

$T \ \rightarrow \ T \ * \ F \ \ | \ T \ / \ F \ | \ \mathtt{F}$

becomes

$$\begin{array}{lcl} T & \rightarrow & F \ T' \\ T' & \rightarrow & * \ F \ T' \ | \ / \ F \ T' \ | \ \epsilon \end{array}$$

Eliminating all left-recursion

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

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

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

Fairly General Algorithm

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

- arrange non-terminals in some order $A_1, \ldots, A_n$.
- 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 $A_i \rightarrow \beta_1\alpha \ | \ \ldots \ | \ \beta_m\alpha$ where current productions for A_j are $A_j \rightarrow \beta_1 \ | \ \ldots \ | \ \beta_m$
  - eliminate any immediate left-recursion in A_i

Example: Replace $A \rightarrow Sc \ | \ d$

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

and then by $\begin{array}{lcl} A & \rightarrow & bcA' \ | \ dA' \\ A' & \rightarrow & acA' \ | \ \epsilon \end{array}$

Left-factoring

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

Change

$V \ \rightarrow \ \alpha\beta \ | \ \alpha\gamma$

to

$$\begin{array}{lcl} V & \rightarrow & \alpha V' \\ V' & \rightarrow & \beta \ | \ \gamma \end{array}$$

Example:

$S \ \rightarrow \ V$ `:= int`
$V \ \rightarrow$ `alpha '[' int ']'` $|$ `alpha`

becomes

$S \ \rightarrow \ V$ `:= int`
$V \ \rightarrow$ `alpha` V'
$V' \ \rightarrow$ `'[' int ']'` $| \ \epsilon$

Transition Diagrams

Used to specify recursive-descent parsers (and other things...)

- Similar to Finite Automata diagrams

- Edges are labeled with **tokens** (take the transition if that token is next input) and **non-terminals** ("call" the named diagram).

Simplifying diagrams leads to better code for recursive-descent parsers.

- Replace recursive calls by loops.

- Substitute "bodies" for "calls."

Expression parsing using recursive-descent

```
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();
}
```